(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
1 February 2001 (01.02.2001)

PCT

(10) International Publication Number
WO 01/08007 A1

(51) International Patent Classification[7]: G06F 9/44

(21) International Application Number: PCT/US00/20069

(22) International Filing Date: 24 July 2000 (24.07.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/145,051	22 July 1999 (22.07.1999)	US
60/212,841	21 June 2000 (21.06.2000)	US

(71) Applicant: PASSAGE SOFTWARE, LLC [US/US]; 4120 Cox Road, Glen Allen, VA 23060 (US).

(72) Inventor: HOWERY, William, D.; 2715 Spinnaker Court, Richmond, VA 23233 (US).

(74) Agents: MCNAMARA, Brian, J. et al.; Foley & Lardner, Suite 500, 3000 K Street, N.W., Washington, DC 20007-5109 (US).

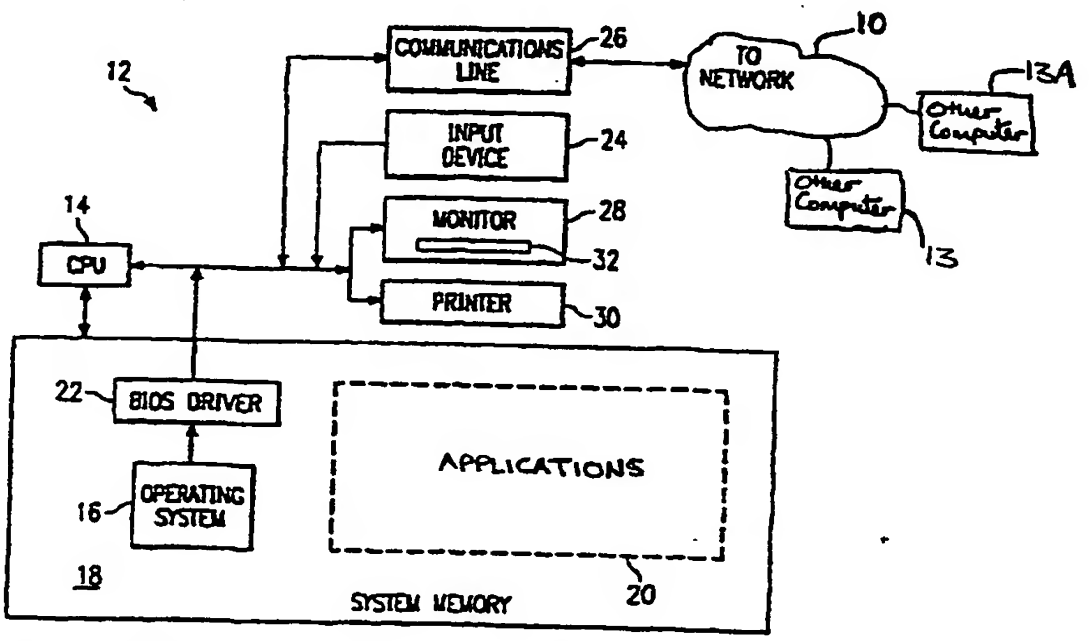
(81) Designated States *(national)*: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published:
— *With international search report.*
— *Before the expiration of the time limit for amending the claims and to be republished in the event of receipt of amendments.*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

(54) Title: METHOD AND SYSTEM OF AUTOMATED GENERATION OF PROGRAM CODE FROM AN OBJECT ORIENTED MODEL



(57) Abstract: A computer implemented method of generating program code from user requirements based modelling diagrams of an object oriented (OO) model including the steps of defining a respective collaboration diagram for each interaction between two objects of the modelling diagram; defining interface, control and entity tier objects based on the modeling diagrams and the collaboration diagram; defining business rule control objects for encapsulating problem domain business rules; defining program flow control objects for encapsulating processes based on interactions; using standardized markup language for communicationg between the business rule control objects and the program flow control objects; and generating program code from the defined interface objects, control objects, entity objects, program flow control objects and business rule control objects.

WO 01/08007 A1

METHOD AND SYSTEM OF AUTOMATED GENERATION OF PROGRAM CODE FROM AN OBJECT ORIENTED MODEL

This application claims the benefit of priority under 35 U.S.C. 119(e) of provisional applications 60/145,051 filed on July 22, 1999, and 60/212,841 filed on June 21, 2000. The contents of both of these provisional applications (including their appendices) are incorporated herein in their entireties.

BACKGROUND OF THE INVENTION

Field of the Invention

This invention relates generally to the field of automatically generated program code and more particularly to a method and system of automated generation of program code for an object oriented model represented in Unified Modeling Language ("UML") notation.

Background of the Related Art

Current object oriented modeling tools provide the components for modeling the object oriented systems. In addition, these tools often provide limited capabilities for generating some aspects of the program code for these object oriented systems that implement the model. However, existing implementation capabilities of these object oriented modeling tools do not provide the complete end-to-end automated program code generation that completely implements the object oriented model.

Furthermore, since existing tools are not capable of generating the end-to-end code to implement a modeled system they are not able to maintain the integrity between the model and the program code. Accordingly, changes at the program code level have to be mapped back to the model in a manual process which gives rise to errors and also causes inconsistency between the model and the implemented code.

For example, the Rational Rose modeling tool ("Rose") provided by Rational Software Corporation, provides basic SQL data access capabilities, including select, insert, update and delete which can be implemented as Visual Basic code. However, the more

complex query functions require the creation of views external to the Rose tool. Coordinating such external data models with the Rose model gives rise to significant challenges and problems.

Furthermore, as discussed above, although Rose provides code generation capabilities for both Visual Basic and C++, considerable amount of hand coding is required to complete the implementation. Rose creates references to associated objects, references and initializes class attributes, declares class methods and implements prototype (stub) code with return data and input parameter types specified. However, the actual program code to complete the methods must still be produced outside of the Rose tool.

Business rules are defined within the Rose system. However, even though the business rules are defined within Rose, no rigorous procedure is provided within Rose to implement these business rules. That is, Rose does not provide a rigorous method for defining business rules so that they can be automatically implemented as program code by the Rose system.

In this context, it should be noted that UML is a general purpose notational language for specifying and visualizing complex software, especially large, object oriented software projects. UML builds on previous notational methods such as Booch, OMT, and OOSE. The UML is rapidly becoming a standard for object-oriented notations under the auspices of the Object Management Group (OMG). UML seeks to provide a common metamodel and a common notation so as to facilitate the development of systems on an architectural scale. Details of the OMG, UML and a Unified Method of system modeling and development based on UML are disclosed on the Internet, for example, at the following URL's www.omg.org and www.omg.org/uml.

As a background to understanding the present invention, a fundamental aspect of object oriented programming is that objects can be organized into classes in a hierarchical fashion and that the objects are interpretable. Classes are abstract generic descriptions of objects and their behaviors. Therefore, a class defines a certain category of methods and data within an object that belongs to that class. Methods comprise the procedures or code that implement the behaviors of the class and operate on the data within the class. Refinement of the methods of a generic class can be implemented by the creation of sub-classes which inherit the behaviors of its parent classes, from which they depend. In addition, the sub classes can have can have behaviors which originate at the sub class or modify the behaviors of its parent classes.

An instance of a class or an object is a specified individual entity that has an observable behavior. That is, an instance is a specific object characterized by having the behaviors defined by its class. Therefore, instances or objects are created and destroyed dynamically while the class is the abstract category under which the instance or object belongs. The instance or object inherits all the methods of its class and its data types but has particular individual values associated with it that are unique. Physically, there is only one location in a computer memory for a class whereas there may be numerous objects or instances of that class each of which has different values and different physical locations in memory.

SUMMARY OF THE INVENTION

Therefore, it is a general object of the invention to alleviate the problems and shortcomings identified above.

One of the objects of the invention is to provide a computer implemented method of generating program code for an object oriented (OO) model in a target environment in which the syntax for the data type primitives is generated and these generated data types are used in generating program code for the OO model.

Another one of the objects of the invention is provide a computer implemented method of generating program code from the modeling diagrams of an OO model.

Another object of the invention is to provide a computer implemented method of depicting asynchronous and synchronous events in a single state diagram of an OO model.

A further object of the invention is to provide a computer implemented method for an OO model in which the program code for a method is synchronized to the activity diagram of that method.

Another object of the invention is to provide a computer implemented method of generating context sensitive help for any element in an activity diagram of an OO method of an object in which context sensitive help is provided based on the context of the object.

These and other objects are achieved by providing a computer implemented method for generating program code for an Object Oriented (OO) model in a target environment including the steps of: defining abstract data type primitives for elements in an activity diagram of an Object Oriented (OO) method; generating the syntax for the abstract data type primitives in the target environment; and generating program code in the target environment, using the generated syntax for the abstract data types, for the OO method

depicted in the activity diagram.

Also provided is a computer implemented method of generating program code from user requirement based modeling diagrams of an object oriented (OO) model including the steps of: defining a collaboration diagram depicting the interaction between two artifacts of the modeling diagrams; defining interface, control and entity tier objects based on the modeling diagrams and the collaboration diagram; defining business rule control objects for encapsulating problem domain business rules; defining program flow control objects for encapsulating processes based on interactions; using standardized markup language for communicating between the business rule control objects and the program flow control objects; and generating program code from the defined interface objects, control objects, entity objects, program flow control objects and business rule control objects.

Also provided is a computer implemented method of depicting asynchronous and synchronous events in a single state diagram of an OO model including the steps of defining a guard condition constraint on an event transition between two states; determining an event transition as being synchronous if a guard condition is present; and determining an event transition as being asynchronous if a guard condition is not present.

Further provided is a computer implemented method of synchronizing program code for an Object Oriented (OO) method to the activity diagram of the OO method in an OO model including the steps of correlating each elements of the activity diagram to at least one line of program code; scanning the program code to identify lines of program code not correlated to any element of the activity diagram; inserting an element in the activity diagram corresponding to respective identified lines of program code not correlated to any elements of the activity diagram.

Also provided is a computer implemented method of generating context sensitive help for any element in an activity diagram of an Object Oriented (OO) method attached to an object in an OO model, including the steps of: determining the element of the activity diagram pointed to by a pointing device; scanning a local namespace available within the object to which the OO method is attached; and generating the context sensitive help based on the element of the activity diagram pointed to and the scanned local namespace of the object.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and constitute a part of the

specification, illustrate a presently preferred embodiment of the invention, and, together with the general description given above and the detailed description of the preferred embodiment given below, serve to explain the principles of the invention.

Fig. 1 is schematic diagram illustrating a computer system suitable for implementing the present invention.

Fig. 2 is a schematic diagram illustrating a computer network that could be used to connect computer systems such as that illustrated in Fig. 1.

Fig. 3 shows a use case diagram for a Logon process use case.

Fig. 4 illustrates a collaboration class object diagram.

Fig. 5 illustrates a collaboration diagram.

Fig. 6 illustrates a control entity class diagram showing peer relationships.

Fig. 7 illustrates a class diagram demonstrating control entity data access showing relationships between database control classes.

Fig 8 illustrates exemplary data access code generated by the present invention.

Fig. 9 shows sample generated Java code implementing a mapping from a logical entity to a physical database.

Figs. 10a and 10b show a sample Java control schema generated according to the present invention.

Fig. 11 illustrates a class diagram for control program flow and control business rule classes.

Fig. 12 illustrates a sample state diagram for VRULogionPF control program flow class object.

Figs 13a-13c show sample code for a generated state machine.

Fig. 14 illustrates an activity diagram for an event/state intersection point.

Fig. 15 shows exemplary code for a generated method corresponding to the scenario diagram of Fig. 14.

Fig. 16 illustrates an exemplary XML object diagram.

Fig. 17 illustrates a generated XML schema.

Fig. 18 illustrates a generated XML DTD.

Figure 19 illustrates a template for a data access.

Figure 20 illustrates a resulting class specification from the merging of a template and a pattern.

Figure 21 shows additional examples of merging templates and patterns to generate

class specifications.

Figures 22-73 disclose the details of one preferred embodiment of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

With reference to the figures, Figure 1 shows a block diagram showing the components of a general purpose computer system 12 connected to an electronic network 10, such as a computer network. The computer network can also be a public network, such as the Internet, a private network or a virtual private network. As shown in the figure 1, the computer system 12 includes a central processing unit (CPU) 14 connected to a system memory 18. The system memory 18 typically contains an operating system 16, a BIOS driver 22, and application programs 20. In addition, the computer system 12 contains input devices 24 such as a mouse and a keyboard 32, and output devices such as a printer 30 and a display monitor 28.

The computer system generally includes a communications interface 26, such as an ethernet card, to communicate to the electronic network 10. Other computer systems 13 and 13A also connect to the electronic network 10 which can be implemented as Wide Area Network (WAN) or as an inter-network such as the Internet. One of skill in the art would recognize that the above system describes the typical components of a computer system connected to an electronic network. It should be appreciated that many other similar configurations are within the abilities of one skilled in the art and all of these configurations could be used with the methods of the present invention. Furthermore, it should be recognized that the computer system and network disclosed herein can be programmed and configured, by one skilled in the art, to implement the method steps discussed further herein.

In addition to being operable in a single computer system, the present invention may also be implemented in a networked computer system. An example of a typical computer network is shown in Fig. 2 which depicts a plurality of workstations 140 connected directly or through a host server 142 to a client/server network 144. The client/server network may include an object oriented messaging system , such as the Object Management Group's Common Object Request Broker (CORBA) or Microsoft's Component Object Model (COM) for controlling the communication between distributed objects across clients and servers. The client/server computers could be connected using a

standard ethernet bus although it is to be understood that all other types of networks, including wireless networks, could also be used in implementations of the present invention.

One aspect of the present invention provides the generation of data access classes from entity and control classes. A prior art modeling tool, such as Rational Rose, provides basic SQL data access capabilities including select, insert, update and delete realized as Visual Basic code. More complex data query functions require the creation of views outside of Rose. Coordinating an external model with the Rose model creates significant challenges. The present invention provides a two-tier data model having an entity tier and a control tier. The entity tier captures the physical and logical relationships. The control tier tracks the relationships between objects. This allows the data modeling to be done independent of business process constraints. The data access functionality can be generated in several programming languages such as Java and Visual Basic. If Visual Basic is the target, joins can be modeled without the creation of separate views. The more sophisticated capabilities of Java allow for the database schema and the table relationships from the control tier to be incorporated directly into the implementation code. A business object model can then use this code to generate complex queries spanning multiple tables.

A preferred embodiment of the present invention will be described next. It should be understood that the following description describes the preferred embodiment of the invention and is not intended to be limitative of the invention. Therefore, the preferred embodiment uses UML terminology and modeling diagrams implemented by Rose to illustrate the preferred embodiment. However, the present invention is intended to apply to all object oriented modeling and development systems that use modeling diagrams and notation terminology that is equivalent to that discussed below with respect to the preferred embodiment.

Fig. 3 shows a use case diagram for a logon process use case. The use case diagram is an example of a use requirement based modeling diagram or construct. In the following description, the "logon" process is described because it is representative of the processes that may be implemented using the present invention. A member of the public (actor) 200 starts a logon process to, for example, access a system that requires a log-in. The logon interaction 205 defines an interaction between the actor and a Logon process 210 that implements the login to the system.

The present invention requires that each interaction must have a collaboration

diagram defining the action and the attributes involved in that action. A collaboration diagram specifies the tangible artifacts defining the interaction between a use case process and an actor or with other processes. Therefore, the preferred embodiment of the present invention requires a one-to-one relationship between the interaction and the collaboration diagrams. In this context, tangible artifacts are defined and measurable pieces of information being sent from a use case process to the actor or from the actor to the use case process. Therefore, artifacts represent the information through to the system.

Creation of a collaboration diagram requires the definition of objects and their attributes within a class diagram. Fig. 4 illustrates a collaboration class object diagram that defines the objects and their attributes that are necessary for the collaboration diagram that is illustrated in Fig. 5. Therefore, Fig. 4 contains exemplary UML class specifications with the Storyboard Stereotype. A Storyboard Stereotype identifies user interface elements. The attributes and attribute type in the class specification define the types of artifacts used with the user interface pages represented by a storyboard stereotype class specification. Therefore, the objects 401, 402, 403, and 404 shown in Fig. 4 correspond to elements 501, 502, 503, and 504, respectively, in the exemplary collaboration diagram shown in Fig. 5

The collaboration diagram illustrated in Fig. 5 depicts the logon interaction 205 identified in the use case diagram shown in Fig. 3. The collaboration diagram produced through this process is tightly coupled to both the use case (shown in Fig. 3) and the objects defined in the class diagram (shown in Fig. 4). Collaboration diagrams contain references place holders for user interface objects. The user interface objects (Storyboards) contain the artifacts or abstract data elements that a user will either send or receive from the system. These events are modeled into a process flow that identifies which additional user interface or logical domain objects will receive the users input. Logical domain objects also respond with result events that trigger transitions to other interface or domain objects within the collaboration diagram.

The next step in the present invention involves data modeling of the persistent and tangible artifacts which are artifacts that are stored in a database or other long term storage. In the prior art, Ivar Jacobson defined three primary system partitioning stereotypes: interface; control; and entity. The entity tier (or stereotype) related to the persistent data objects, the control tier to the mid-tier processes, and the interface tier to the interactions with the users or other systems. The present invention provides two

additional tiers: a control program flow tier and a control business rules tier. Both of these additional tiers provide additional definition to the mid-tier processing and are discussed in more detail further herein.

Therefore, the present invention provides that two class diagrams are developed: (i) an Entity Relationship Diagram (ERD) class diagram defining the entity and the control tiers (similar to Jacobson's Objectory process); and (ii) a control program flow/business rules class diagram. Fig. 6 shows an exemplary control entity class diagram showing peer relationship between two objects, 601 and 602.

The present invention provides that once the control stereotype objects and their relationships are defined, as discussed above, the peer relationships and the entity models for the Entity Relationship Diagrams are automatically generated. Therefore, the present invention provides that the program code implementing the invention understands the relationship between the control data access and the entity persistent data access classes. When this relationship is created the program code automatically sets the relationships required for generating the complete program implementation for the control and entity data access. This automatic generation provides consistency, completeness, and reduces a significant amount of tedious and redundant effort in prior art systems.

The present invention also generates a class diagram demonstrating control entity data access showing the relationships between Database control classes as shown, for example, in Fig. 7. Fig. 7 illustrates an exemplary set of data access classes 701-705 with their interconnecting relationships. The present invention understands the attributes of the data access classes through the schema information extracted from the UML model in the entity and control class specifications. The invention provides for propagating the role association names identifying the data elements of the control data access specifications that relate each class to other classes. The present invention also creates program code operation methods for implementing run-time navigation between control data access class specifications.

The present invention then provides for creating the code for data access. A generator processes each entity class using the properties as defined by the processes discussed above. It creates a metastructure defining a mapping between class and class attributes to database name and columns. To define this mapping , the present invention provides for using the semantic naming applied to the UML model. With this semantic meaning, the present invention understands how each attribute of the entity class

specification will be implemented into a persistent data base environment. This semantic information is interpreted in program code information which automates how all objects are manipulated in run-time programs. This mapping is generally Interface Definition Language (IDL) specific. This process implements an interface (presently in Java and COM) that the business database object model can use to interpret the class diagrams for managing data access. Exemplary Java program code that implements select/update/delete and select loading of a collection is shown in Fig. 8.

The present invention then generates the control stereotype tier. This is done by cycling through the class diagram, picking out each class stereotype control and associated peer-relationship entity class. The control class uses the related entity class to perform data access management. Additionally, the database schema diagram is abstracted from the class diagram. This schema is then used to dynamically create complex query joins between multiple objects. Fig. 9 shows sample generated Java code implementing a mapping from a logical entity to a physical database. Figs. 10a-10b show a sample Java control schema generated in accordance with the present invention. The present invention uses the UML model that contains the relationships between persistent class specifications and also contains the information required for these specific class objects to access and manipulate information within a database. When the present invention creates the program code to access and manipulate information within a database, it interprets the UML model relationship information creating a schema or schematic representation of the UML model. This schema contains the knowledge required for the present invention to automatically transform relationship database information into object oriented run-time program structures.

The next step in the present invention requires the definition of the business rules. In the collaboration diagram illustrated in Fig. 5, a submit action 505 is shown that activates a business rule. A business rule is a process that evaluates business domain data and makes a decision accordingly. The present invention provides that business rules are defined under two separate stereotypes, either as control program flow or a control business rule stereotypes. The control business rules encapsulate the problem domain business rules. The control program flow objects control the path a transaction takes through the application based on the constraints provided by the business rules. These control program flow and control business rules can be defined in a class diagram. Each class contains a state diagram which defines events, states, and methods that belong to that

class.

Fig. 11 shows a class diagram for exemplary control program flow and control business rule classes 1101 and 1102, respectively. In addition, it shows the relationships between the control business rules and the database control classes. Fig. 12 shows a representative state machine. Therefore, each control process flow for control business rule object may have an associated state chart diagram. The present invention provides for translating this state chart diagram to a program code representation to manage events coming into the object or transitioning through the various states of the object. Figure 11 also illustrates a representative set of relationships between associated objects completing a system implementation. Typically, a control process flow object will relate to some domain business rule object. That control business rule object will then subsequently relate to database elements through the control data access stereotyped class specification.

Once the class diagrams and the state machines have been defined as discussed above, the present invention provides that the class code embodying the state machine that executes the business rule is generated. A state diagram may be associated with any control process flow or control business rule class specification. The state chart contains events transitioning between the defined states of that object. The present invention translates between these events and the guard conditions navigating boolean transitions between states into a code implementation that can receive synchronous and asynchronous events to implement the state transitions for a given object. Sample code for a generated state machine (shown in Fig. 12) for the VRULogonPF control program flow object is shown in Figs. 13a-13c.

It should be noted that the implementation code for class, business rules, and program flow can be used to generate standard container patterns (set dictionary lookup, hashtable, etc.) for managing associated objects. The abstraction of relationships to other objects facilitates the access and maintainability of complex object models.

In order to generate the class code for a state machine to function in a completed application, the present invention requires that the specific operations that take place at each intersection point between an event and a state be defined. This is accomplished though the use of activity diagrams, one of which must be associated with each event/state intersection point. Fig. 14 provides an example of an activity diagram for one of the intersection points between an event and a state shown in the state diagram of Fig. 12. This activity diagram corresponds to the intersection between the event = "TooMany" and

state = "Fail."

Fig. 15 shows exemplary generated code for the method of the VRULogonPF activity diagram (as shown in Fig. 14) for the event = "TooMany" and state = "Fail."

Therefore, the present invention extends beyond the conventional use of an activity diagram in a conventional tool, such as that provided by Rose, implementing UML. In the prior art, activity diagrams are used to graphically represent decisions and flows through an application. However, the prior art did not provide for the ability to define a behavioral scope of the activity diagrams, or simplify complex algorithms with many steps into their block representations of the activity diagrams. Furthermore, the prior art was unable to minimize detail required for representing swim lane objects in the activity. The present invention provides for assigning behavioral scope to the activity diagrams and provides for collapsing multiple complex steps into single activities. The present invention provides program code to automate work for a designer to manage multiple swim lanes and interactions between multiple swim lanes.

Therefore, one aspect of the present invention provides a specific one-to-one correlation between the event states, the class operations and activity diagrams. Therefore, by combining the business rules, modeled as described above, and the state machine activity diagrams, the present invention permits the generation of complete program code from an object model. Furthermore, another benefit of the present invention is that the information required for code generation is modeled in a specific sequence of modeling constructs and the program code can be generated from these modeling constructs in any target platform/language combination that supports the modeling constructs described above. Therefore, the modeling constructs are not specific to any single platform or language.

Another aspect of the present invention provides that cross tier communications can be accomplished by using the constructs of standardized markup language, such as XML (eXtensible Markup Language). Therefore, the present invention provides that the tiers of the completed application communicate through XML. This creates the challenge of managing free-form text information in a structured object oriented paradigm. The present invention addresses this challenge by extending UML onto the XML world. This is done by providing XML map extensions to the class diagrams. Therefore, the present invention provides that the XML stereotype classes and associations are set and the relationships between them are defined. Fig. 16 provides an examples of an XML object diagram

according to the present invention.

These XML maps are then related to a control program flow object as shown in Fig. 16. Therefore, when the present invention generates the application, it interprets the associated XML diagram to generate the Data Type Definition (DTD) and the XML schema used by the business object model. Fig. 17 provides an exemplary XML schema generated to correspond to the XML object diagram shown in Fig. 16. Fig. 18 shows the exemplary DTD generated from the XML map object diagram shown in Fig. 16.

Therefore, one aspect of the present invention provides for understanding and using the relationships between multiple objects in UML class diagrams. The invention then transforms these relationships into structural affirmations with schema information similar to database access representation. This structural schema information provides the information for program code (according to the present invention) to insert, extract, and navigate through XML documents. The structure of these XML documents can be defined through the UML model. The invention uses the XMLMap stereotype to represent a hierarchical structure for transformation into an XML map schema. The control process flow attached to the XML map structure receives the program code to define and implement the XMLMap manipulation and navigation program code.

One aspect of the present invention provides a method of synchronization of the program code that implements a method and an activity diagram that corresponds to that method. This is accomplished by correlating each element of the activity diagram to one or more lines of program code that implement the method. Furthermore, the sequence of elements in the activity diagram are also correlated to a sequence of the identified lines of program code that correspond to a particular elements.

Accordingly, if the program code is changed the method of the present invention provides that the program code is scanned to determine if any of lines of program code are not correlated to an element of the activity diagram. If any lines of uncorrelated program code are found, an activity diagram element that corresponds to the uncorrelated program code is inserted in the activity diagram. Since, the sequence of the elements of the activity diagram are also correlated to the sequence of the lines of program code, the position of the uncorrelated lines of code is used to determine the insertion point of the element in the activity diagram. In this way, the present invention synchronizes the program code that implements a method with the activity diagram corresponding to that method.

Another aspect of the present invention provides a computer implemented method of

generating context sensitive help for any element in an activity diagram of an Object Oriented (OO) method attached to an object in an OO model. The method involves determining the element of the activity diagram pointed to by a pointing device such as a cursor controlled by a mouse. Thereafter, the method of the present invention scans a local namespace available within the object to which the OO method is attached, and generates the context sensitive help based on the element of the activity diagram pointed to and the scanned local namespace of the object. This context sensitive help also builds the UML constructs that permit the generation of the program code that implements a method corresponding to an activity diagram.

A further aspect of the present invention provides a computer implemented method for generating the target code for data types in various different target environments. To this end, the present invention provides for the definition of abstract data type primitives for the data types. For example, these abstract data type primitives can be implemented as objects with data and methods (or behavior) corresponding to a particular data type. The present invention also provides for the generation of target code for these abstract data type primitives for a particular target environment. In this way, the present invention provides for modeling business objects and data types independent of the target environment.

Another aspect of the present invention provides a computer implemented method for depicting both synchronous and asynchronous events in a singe state diagram of an OO model. This is accomplished by interpreting an event having a guard condition constraint as defining a synchronous event transition whereas events that do not have guard condition constraints are defined as asynchronous event transitions. For example, with reference to Fig. 12, the guarded events 1201 and 1202 define synchronous event transitions while the event transitions 1203 and 1204 that do not have a guard condition define asynchronous event transitions. In this way, the present invention provides for the representation and interpretation of both synchronous and asynchronous events in one state diagram.

Another aspect of the present invention provides a computer implemented method and software for using template and pattern classes to generate a full class specification. Accordingly, the present invention provides that a class specification and activity details are generated from a tokenized UML template. Many software modules have precise patterns for implementation. These patterns have variations that are specific to particular instances of a given application.

In this context, it should be noted that in object-oriented programming, a pattern

can contain the description of certain objects and object classes to be used, along with their attributes and dependencies, and the general approach for solving a particular problem. A collection of patterns, called a pattern framework, can also be used to solve a specific problem. A book, "Design Patterns: Elements of Reusable Object-Oriented Software," by E. Gamma, R. Helm. R. Johnson, and J. Vlissides is crediting with creating an interest in design patterns in object-oriented programming.

The problem of database access, for example, implies a very rigid pattern on manipulating the database tables and fields. The variant components for a given class on a database access pattern, for example, would be the number of data columns and class attributes, as well as the names and types of these attributes. The data access pattern could utilize the same processes and procedures for reading, writing, creating and updating the class table object. The present invention provides for modeling the precise and constant elements of a design pattern, and for substituting at design time the variant elements of that particular design pattern. By utilizing template definitions within the class specifications allows the present invention to iterate a template specification over a specific instance that defines the variant elements of the pattern.

The present invention tokenizes the various static elements of a class specification and creates a new class specification merged from a template and a pattern. The template contains tokenized names identifying elements such as Class Name, Attribute Name, Association, operation and other static class items. The present invention provides a <<Pattern>> Stereotype Class Specification that contains the definitions for these variant token elements from the present invention's <<Template>> Stereotype Specification.

By utilizing a template Stereotype Class Specification and multiple Patterns for the variant elements, the present invention creates a complete Class Specification, with static and dynamic elements implemented per the Pattern associations within the variant template stereotype Class Specification.

Figure 19 is an example that demonstrates a Template 1901 for an EJB (Enterprise Java Bean) data access. The template 1901 is realized by a Pattern 1902, specifying the Class Name, Attributes, and Method names. The resulting class 2001 (shown in Fig. 20) is generated by the merging of the Template 1901 and the Pattern 1902.

As shown in figure 20, each tokenized element within the pattern specification is replicated for the template specification for each item defined in the pattern tokens. For

example: the attribute token is defined for the pattern set and get methods. According to the present invention, this instructs the present invention to create a get and set method for each attribute in the target template. The target template 2001 will define variant attributes and Data types for those attributes. The pattern specification will define a prototypical get and set method for attributes and the complete activity implementations for the attribute methods. When the present invention generates the complete class specification for this pattern and template relationship, it will substitute each attribute name and data type into the respective tokens for the prototypical methods defined within the pattern. The result is a complete class with the get and set methods for each attribute in the template. That is, the prototypical behavior is defined within the template and transferred to the pattern for each tokenized element. The pattern contains the data elements and/or methods with tokenized attributes or class definitions. When the template is applied to the pattern the dynamic behavior of the template will be transformed for each token element within the pattern.

Figure 21 is another example in which templates 2101 and 2102 are merged with one or more of the patterns 2103-2105 to generate class specifications 2106-2109. That is, class specification 2106 is generated by merging template 2102 with pattern 2103, class specification 2107 is generated by merging template 2102 with pattern 2104, class specification 2108 is generated by merging template 2101 with pattern 2105, and class specification 2109 is generated by merging template 2102 with pattern 2105.

According to the present invention, this pattern process can be implemented for any predictive and repetitive software pattern. A designer creates the predictive pattern with tokens representing the variant elements and then creates a template defining the variant elements. Then, the present invention provides for merging the pattern and template to create the complete class specification.

Figs. 22-73 disclose the details of one preferred embodiment of the present invention.

Other embodiments of the invention will be apparent to those skilled in the art from a consideration of the specification and the practice of the invention disclosed herein. It is intended that the specification be considered as exemplary only, with the true scope and spirit of the invention being indicated by the following claims.

What is claimed is:

1. A computer implemented method of generating program code for an Object Oriented (OO) model in a target environment comprising the steps of:

defining abstract data type primitives for elements in an activity diagram of an Object Oriented (OO) method;

generating the syntax for the abstract data type primitives in the target environment; and

generating program code in the target environment, using the generated syntax for the abstract data types, for the OO method depicted in the activity diagram.

2. A computer implemented method of generating program code from user requirements based modeling diagrams of an object oriented (OO) model comprising the steps of:

defining a respective collaboration diagram for each interaction between two objects of the modeling diagram;

defining interface, control and entity tier objects based on the modeling diagrams and the collaboration diagram;

defining business rule control objects for encapsulating problem domain business rules;

defining program flow control objects for encapsulating processes based on interactions;

using standardized markup language for communicating between the business rule control objects and the program flow control objects; and

generating program code from the defined interface objects, control objects, entity objects, program flow control objects and business rule control objects.

3. A computer implemented method of depicting asynchronous and synchronous events in a single state diagram of an OO model comprising the steps of:

defining a guard condition constraint on an event transition between two states;

determining an event transition as being synchronous if a guard condition is present; and

determining an event transition as being asynchronous if a guard condition is not

present.

4. A computer implemented method of synchronizing program code for an Object Oriented (OO) method to the activity diagram of the OO method in an OO model comprising the steps of:

correlating each elements of the activity diagram to at least one line of program code;

scanning the program code to identify lines of program code not correlated to any element of the activity diagram; and

inserting an element in the activity diagram corresponding to respective identified lines of program code not correlated to any elements of the activity diagram.

5. A computer implemented method of generating context sensitive help for any element in an activity diagram of an Object Oriented (OO) method attached to an object in an OO model, comprising the steps of:

determining the element of the activity diagram pointed to by a pointing device;

scanning a local namespace available within the object to which the OO method is attached; and

generating the context sensitive help based on the element of the activity diagram pointed to and the scanned local namespace of the object.

6. A computer readable data storage medium having program code recorded thereon for generating target program code for an object oriented model in a target environment, the program code comprising:

a first program code that allows the definition of abstract data type primitives for elements in an activity diagram of an object oriented (OO) method;

a second program code that generates the syntax for the abstract data type primitives in the target environment; and

a third program code that generates the target program code in the target environment, using the generated syntax for the abstract data types, for the OO method depicted in the activity diagram.

7 A computer readable data storage medium having program code recorded thereon for generating target program code from user requirements based modeling diagrams of an object oriented model, the program code comprising:

a first program code that allows the definition of a respective collaboration diagram for each interaction between two objects of the modeling diagram;

a second program code that allows the definition of interface, control, and entity tier objects based on the modeling diagram and the collaboration diagram;

a third program code that allows the definition of business rule control objects that encapsulate the problem domain business rules;

a fourth program code that allows the definition of program flow control objects for encapsulating process based on interaction;

a fifth program code that uses standardized markup language for communicating between the business rule control objects and the program flow control objects; and

a sixth program code that generates the target program code from the defined interface objects, control objects, entity objects, program flow control objects, and business rule control objects.

8. A computer readable data storage having program code recorded thereon for depicting asynchronous and synchronous events in a single state diagram of an OO model method, the program code comprising:

a first program code that allows defining a guard condition constraint on an event transition between two states; and

a second program code that determines the event transition to be synchronous of the guard condition is present and determines the event transition to be asynchronous if the guard condition is not present.

9. A computer readable data storage having program code recorded thereon for synchronizing generated program code for an object oriented (OO) method to an activity diagram of the OO method in an OO model, the program code comprising:

a first program code that correlates each element of the activity diagram to at least one line of program code;

a second program code that scans the generated program code to identify lines of generated program code that are not correlated to any element of the activity diagram; and

a third program code that inserts an element in the activity diagram that corresponds to respective identified lines of program code not correlated to any elements of the activity diagram.

10. A computer readable data storage having program code recorded thereon for generating context sensitive help for any element in an activity diagram of an object oriented (OO) method attached to an object in an OO model, the program code comprising:

a first program code that determines the element of the activity diagram pointed to by a pointing device;

a second program code that scans the local namespace available within the object to which the OO method is attached; and

a third program code that generates the context sensitive help based on the element of the activity diagram pointed to and the scanned local namespace of the object.

11. The method according to claim 2, wherein the step of using standardized markup language to communicate between objects uses the eXtensible markup language (XML) and includes the steps of:

developing an XML map extension to a class diagram to create an XML map object diagram that relates XML map objects to control program flow objects;

generating XML schema from the XML map object diagram; and

generating XML Data Type Definitions (DTD) from the XML map object diagram.

12. A computer implemented method of generating an object oriented class specification, the method comprising the steps of:

creating a template stereotype specification with tokens identifying variant class elements;

creating a pattern stereotype specification containing definitions for the variant class elements; and

merging the template stereotype specification and the pattern stereotype specification to generate a class specification wherein the tokens identifying the variant class elements are replaced by definitions for the variant class elements contained in the pattern stereotype specification.

13. The method according to claim 12, wherein the creating a pattern stereotype specification step includes creating a plurality of pattern stereotype specifications, and

wherein the merging step includes merging any one of the plurality of pattern stereotype specifications with the template stereotype specification to generate the class specification.

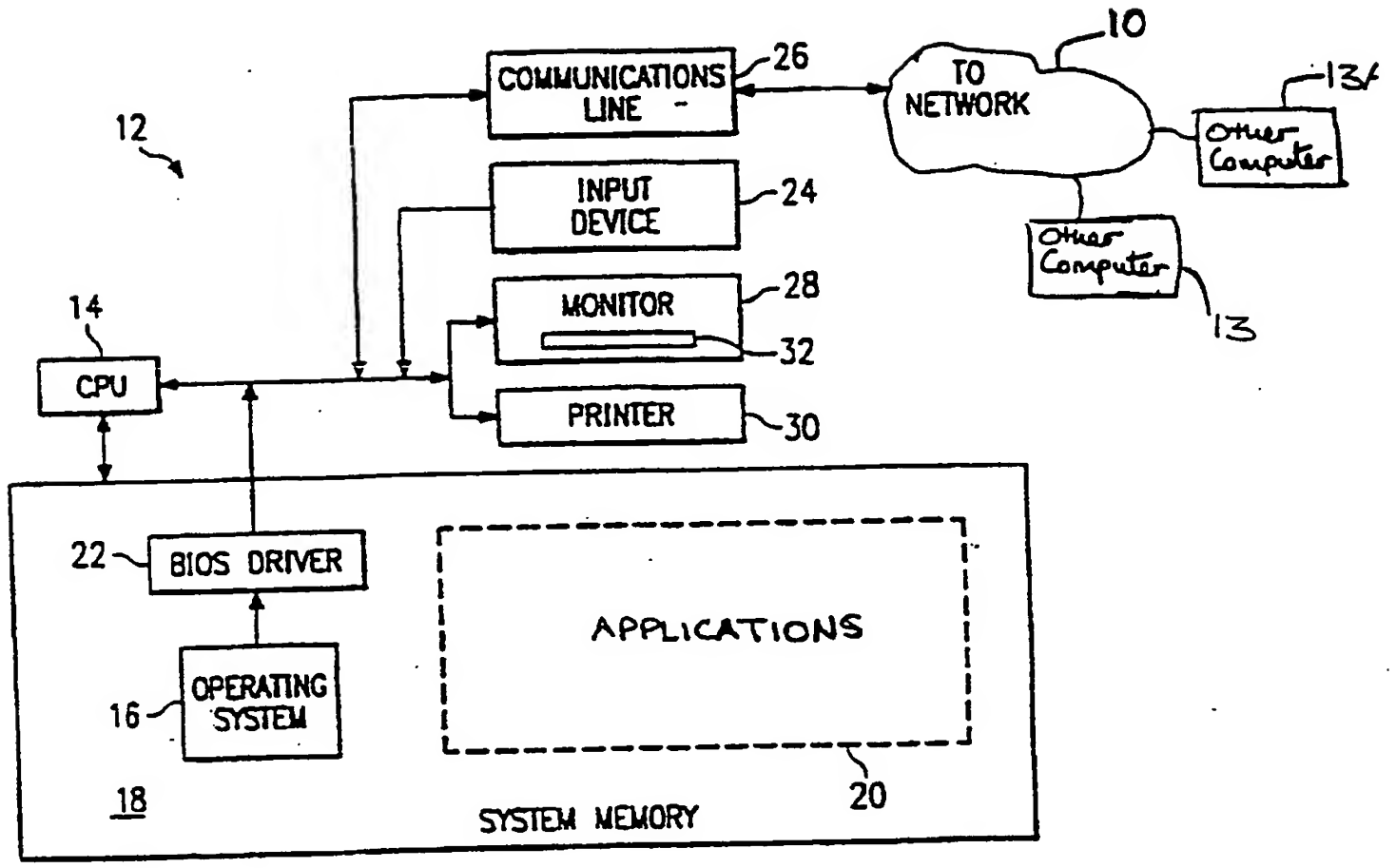

10
12
COMMUNICATIONS LINE
26
TO NETWORK
13A
Other Computer
INPUT DEVICE
24
Other Computer
13
MONITOR
28
32
14
CPU
PRINTER
30
22
BIOS DRIVER
APPLICATIONS
16
OPERATING SYSTEM
18
20
SYSTEM MEMORY

FIGURE 1.

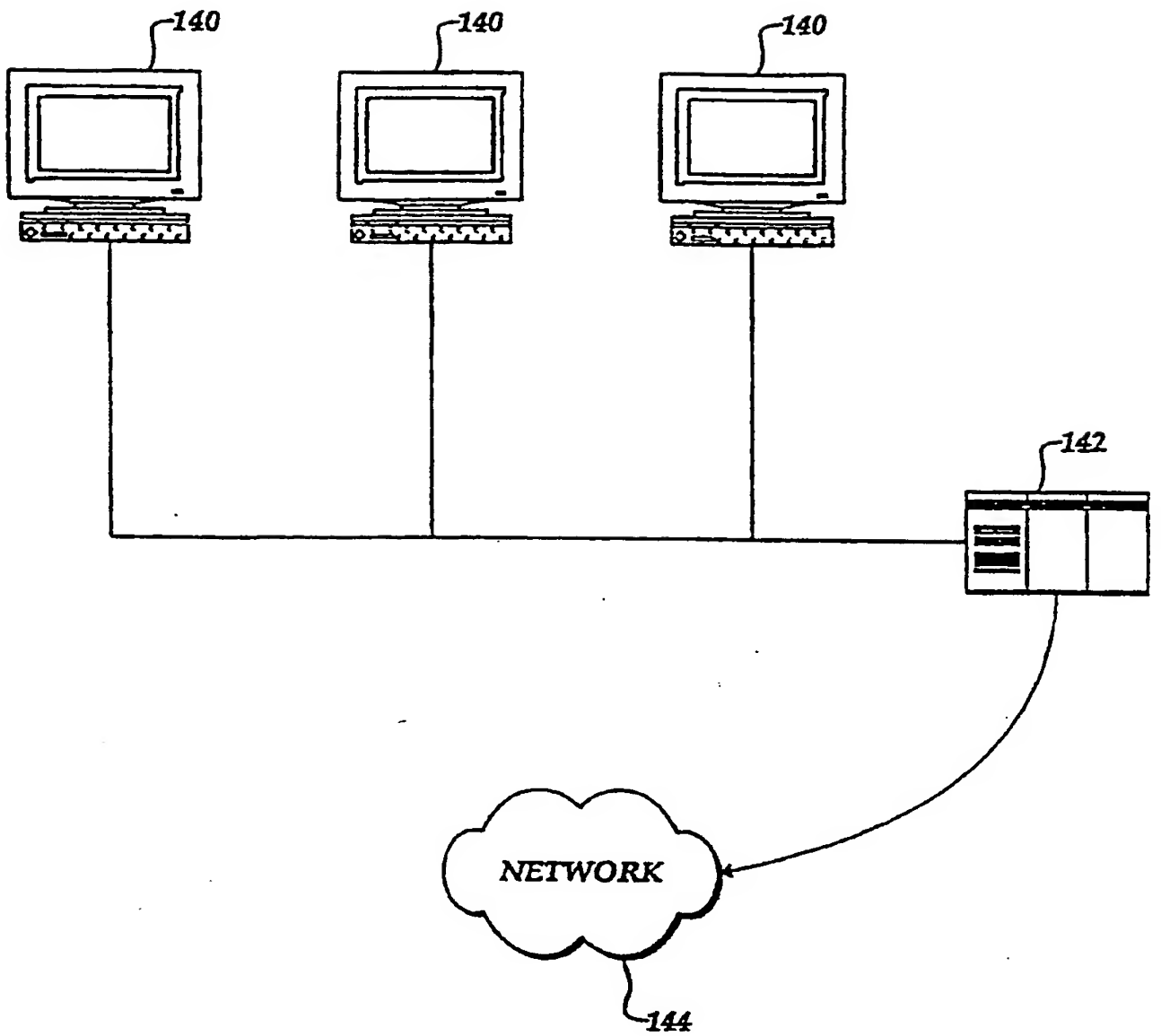


FIG. 2

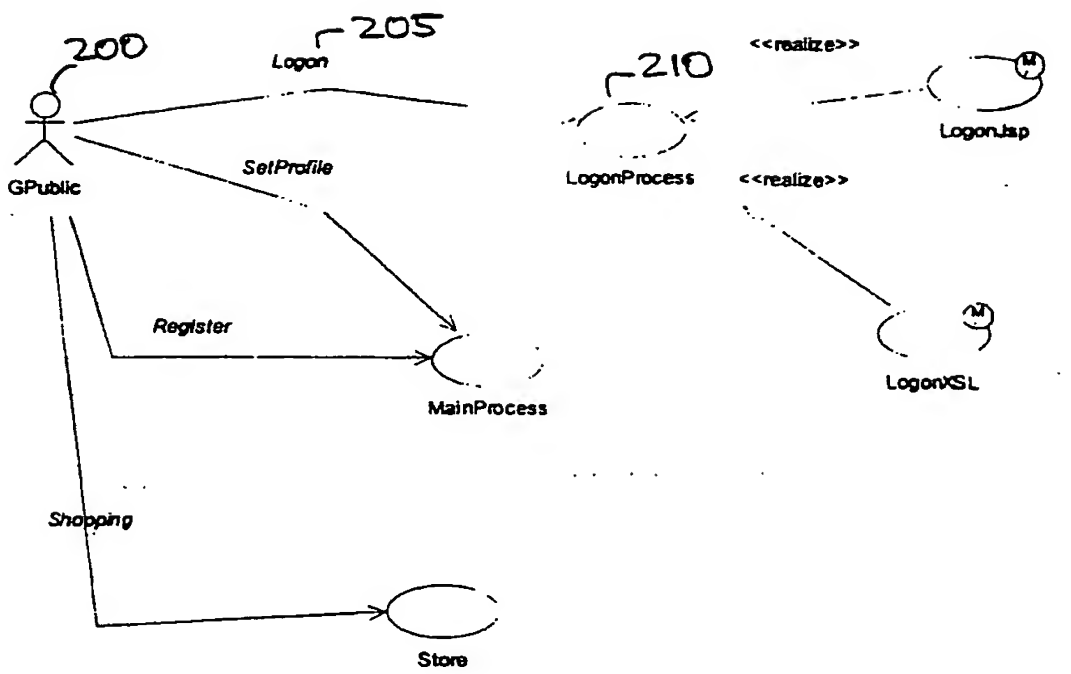

200
205
Logon
210
<<realize>>
LogonJsp
GPublic
SetProfile
LogonProcess
<<realize>>
Register
MainProcess
LogonXSL
Shopping
Store

Figure 3

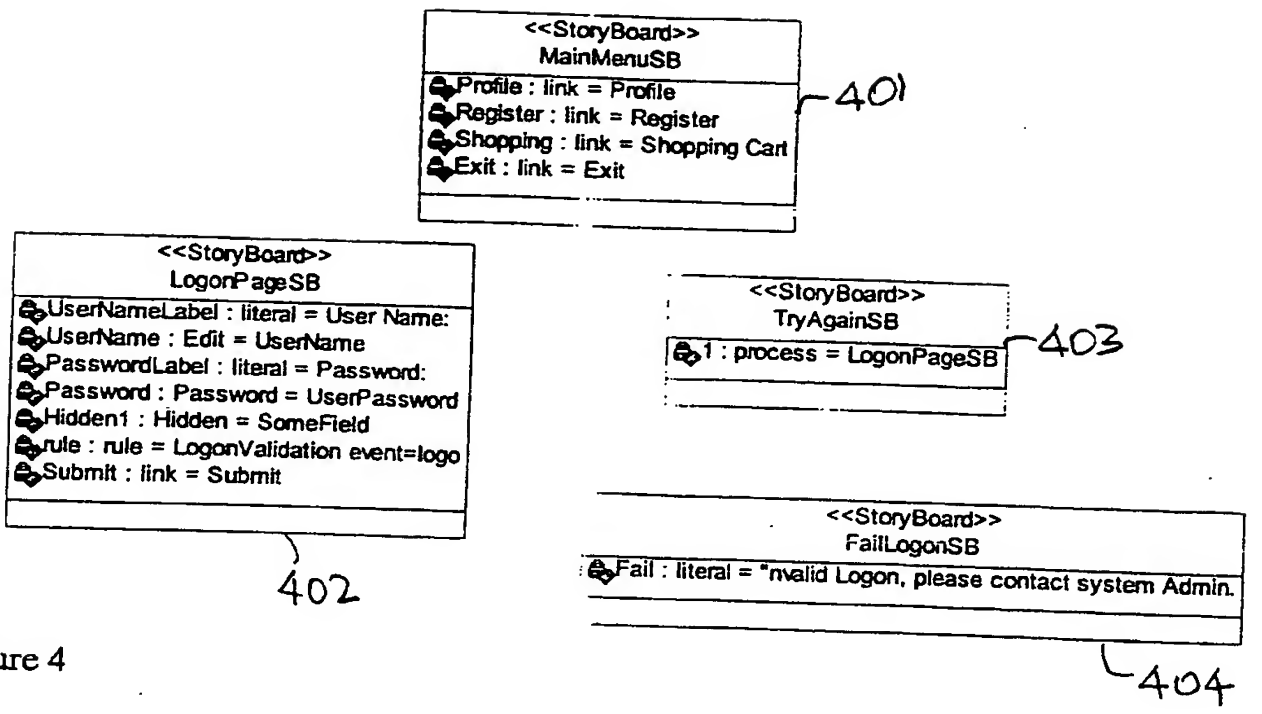

<<StoryBoard>>
MainMenuSB
Profile : link = Profile
Register : link = Register
Shopping : link = Shopping Cart
Exit : link = Exit
401
<<StoryBoard>>
LogonPageSB
UserNameLabel : literal = User Name:
UserName : Edit = UserName
PasswordLabel : literal = Password:
Password : Password = UserPassword
Hidden1 : Hidden = SomeField
rule : rule = LogonValidation event=logo
Submit : link = Submit
402
<<StoryBoard>>
TryAgainSB
1 : process = LogonPageSB
403
<<StoryBoard>>
FailLogonSB
Fail : literal = "nvalid Logon, please contact system Admin.
404

Figure 4

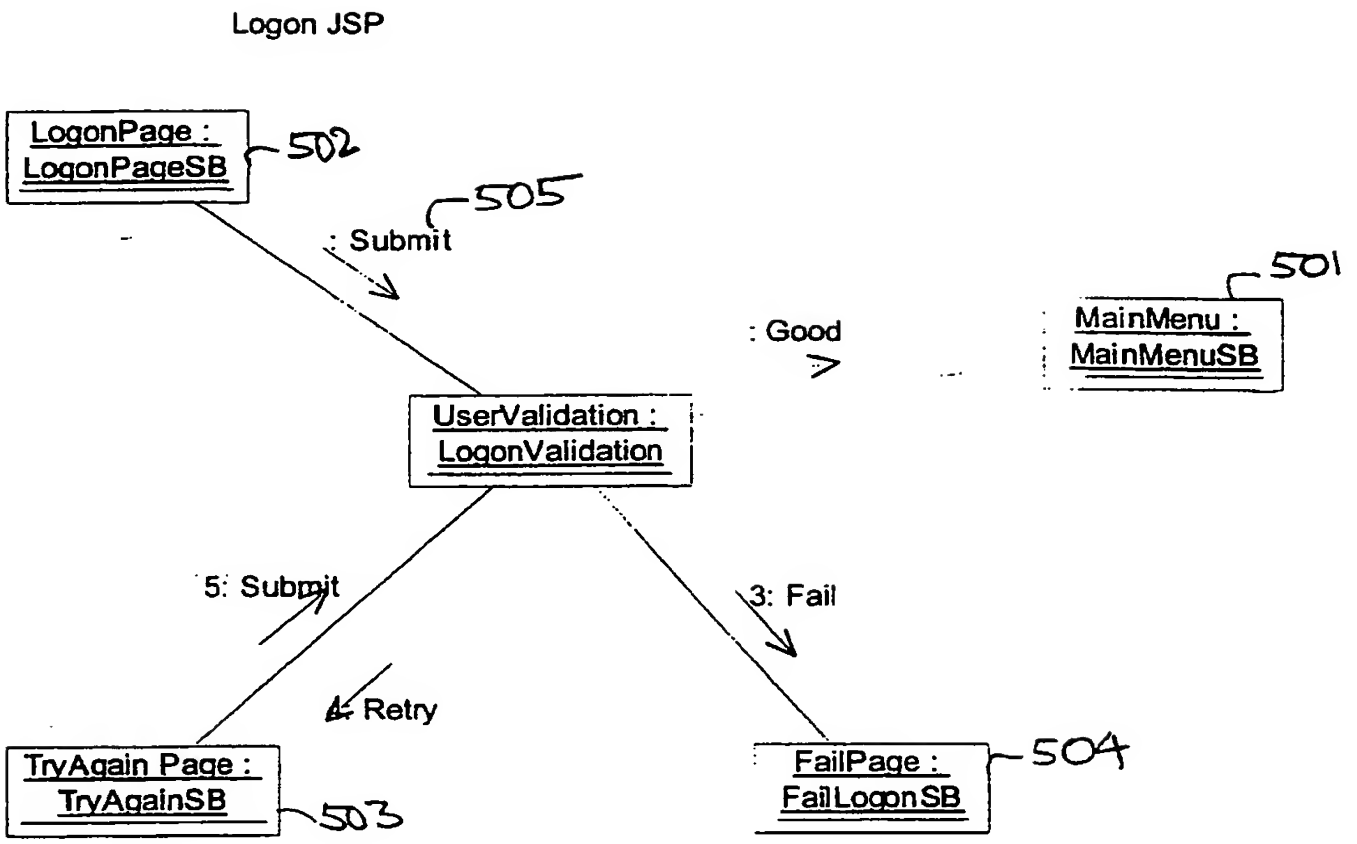

Logon JSP
LogonPage : LogonPageSB
502
505
: Submit
501
MainMenu : MainMenuSB
: Good
UserValidation : LogonValidation
5: Submit
3: Fail
Retry
TryAgain Page : TryAgainSB
503
FailPage : FailLogonSB
504

Figure 5

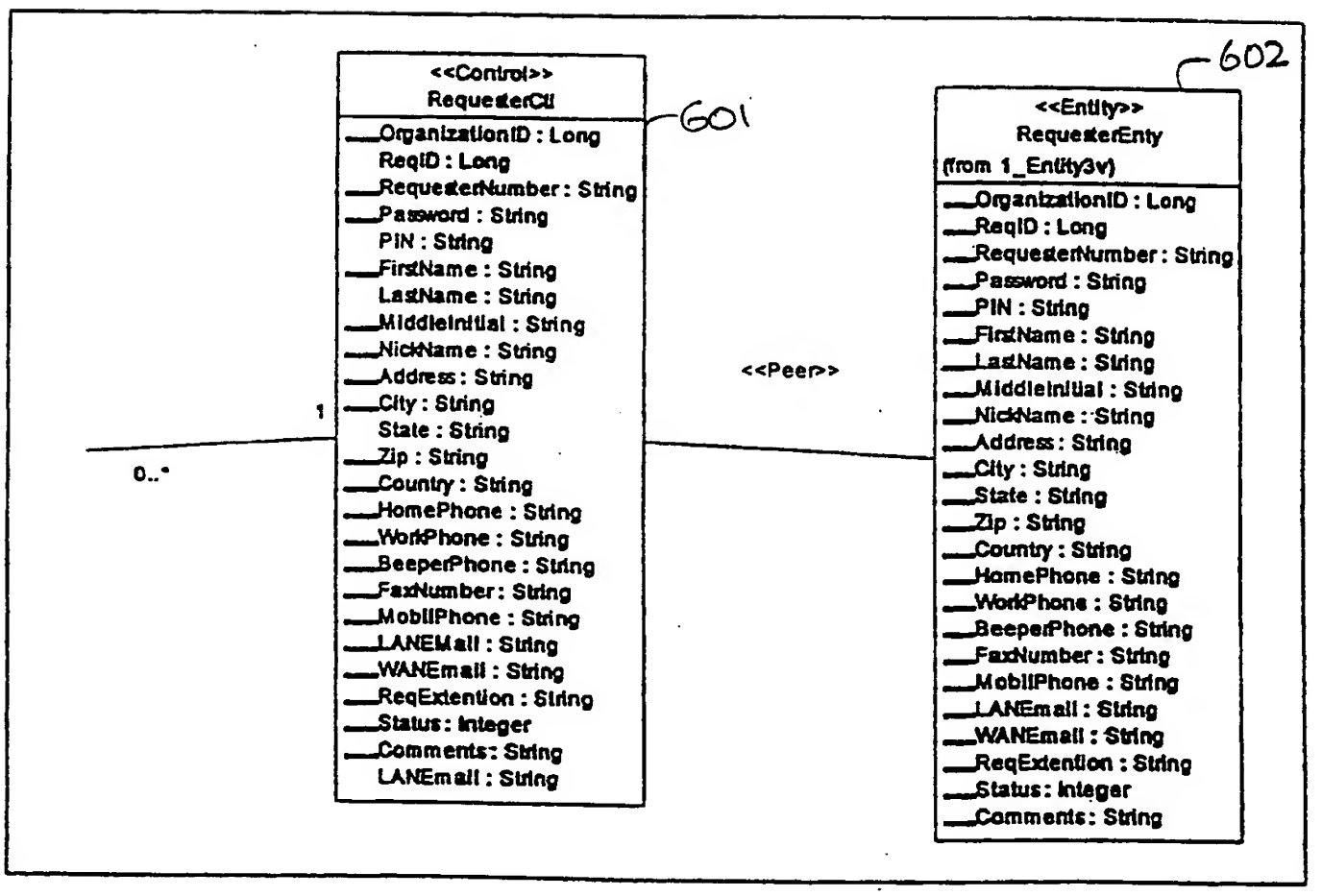

601
602
<<Control>>
RequesterCtl
OrganizationID : Long
ReqID : Long
RequesterNumber : String
Password : String
PIN : String
FirstName : String
LastName : String
MiddleInitial : String
NickName : String
Address : String
City : String
State : String
Zip : String
Country : String
HomePhone : String
WorkPhone : String
BeeperPhone : String
FaxNumber : String
MobilPhone : String
LANEMail : String
WANEmail : String
ReqExtention : String
Status : Integer
Comments : String
LANEmail : String
1
0..*
<<Peer>>
<<Entity>>
RequesterEnty
(from 1_Entity3v)
OrganizationID : Long
ReqID : Long
RequesterNumber : String
Password : String
PIN : String
FirstName : String
LastName : String
MiddleInitial : String
NickName : String
Address : String
City : String
State : String
Zip : String
Country : String
HomePhone : String
WorkPhone : String
BeeperPhone : String
FaxNumber : String
MobilPhone : String
LANEmail : String
WANEmail : String
ReqExtention : String
Status : Integer
Comments : String

FIG. 6

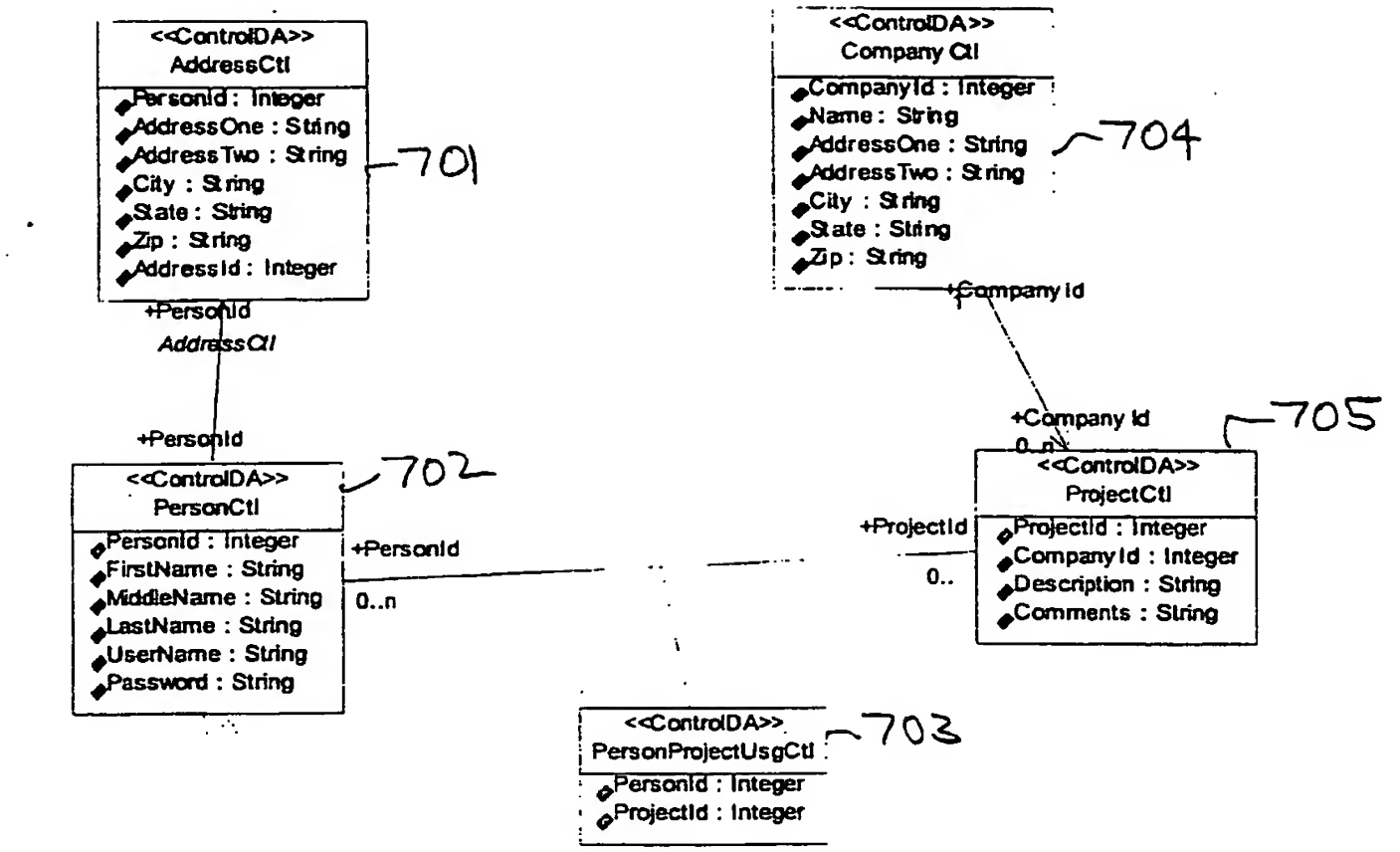

<<ControlDA>>
AddressCtl
PersonId : Integer
AddressOne : String
AddressTwo : String
City : String
State : String
Zip : String
AddressId : Integer
701
+PersonId
AddressCtl
+PersonId
<<ControlDA>>
PersonCtl
PersonId : Integer
FirstName : String
MiddleName : String
LastName : String
UserName : String
Password : String
702
+PersonId
0..n
<<ControlDA>>
PersonProjectUsgCtl
PersonId : Integer
ProjectId : Integer
703
<<ControlDA>>
Company Ctl
CompanyId : Integer
Name : String
AddressOne : String
AddressTwo : String
City : String
State : String
Zip : String
704
+CompanyId
+Company Id
0..n
<<ControlDA>>
ProjectCtl
ProjectId : Integer
CompanyId : Integer
Description : String
Comments : String
705
+ProjectId
0..

Figure 7

```
public class PersonEnty implements Loadable, Cloneable{
                                                // Loadable interface provides access for DBPackage
      DBPackage mDBPackage=null;                // the Database connection this object belongs to
      DBControl mDBControl=null;                // the Control this Entity belongs to
      static DBTableDefinition mDBTableDefinition=null;                    // Static field deffinitions
      boolean    DBPackageConnectionActive=false; // this flag determines the access is coming from the DBPackage



      /****** Start Loadable interface ************/
      public DBTableDefinition getFields(){
         // this Loadable interface provides the DBPackage with the field definitions
         // required for object and field loading
         if (mDBTableDefinition != null){
            return mDBTableDefinition ;
         }
         DBTableDefinition tdef = new DBTableDefinition();
         mDBTableDefinition = tdef;
         tdef.setTableName(mTablePersonEnty);  // Database table name
         tdef.addField("PersonId", mfldPersonId, DBPackage.dbp_Integer,  0, DBPackage.PRIMARY | DBPackage.UNIQUE |
         tdef.addField("FirstName", mfldFirstName, DBPackage.dbp_String,  50,  0);
         tdef.addField("MiddleName", mfldMiddleName, DBPackage.dbp_String,  50,  0);
         tdef.addField("LastName", mfldLastName, DBPackage.dbp_String,  50,  0);
         tdef.addField("UserName", mfldUserName, DBPackage.dbp_String,  50,  0);
         tdef.addField("Password", mfldPassword, DBPackage.dbp_String,  50,  0);
         tdef.setTimeOut( 0);
         RemoteEngineConfig rc = new RemoteEngineConfig();
         long irefresh = rc.getRemoteEngineCacheTimeOut("PersonEnty");
         if(irefresh>0){
            tdef.setTimeOut(irefresh);
         } // if(tdef.getTimeOut()>0){
         tdef.setCacheMe(false);
         return tdef;
      }
```

FIGURE 8

```
public DBSchema getSchema(){
   if (mDBSchema != null){
      return mDBSchema;
   }
   mDBSchema = new DBSchema();
   mDBSchema.setControlName("PersonCtl");
   AssociatedControl assoc =null;
   RelationField rf=null;

   assoc = new AssociatedControl();
   assoc.setName("AddressCtl");
   assoc.setCardinality( new Integer(0));
   mDBSchema.addAssociation(assoc);
   rf = new RelationField();
   rf.setName("PersonId");
   assoc.addFields(rf);

   assoc = new AssociatedControl();
   assoc.setName("PersonProjectUsgCtl");
   assoc.setCardinality( new Integer(1));
   mDBSchema.addAssociation(assoc);
   rf = new RelationField();
   rf.setName("PersonId");
   assoc.addFields(rf);

   return mDBSchema;
}
```

Figure 9

```
OrganizationCtl aOrganizationCtl = new Organization
aOrganizationCtl.setOrgID(aRequesterCtl.getOrganizationID());
GroupingItemCtl aGroupingItemsUsageCtl1 = new GroupingItemCtl();
GroupingItemCtl aGroupingItemsUsageCtl2 = new GroupingItemCtl();
aGroupingItemsUsageCtl1.setObjectType(new Double(1));

aGroupingItemsUsageCtl1.setObjectID(aRequesterCtl.getOrganizationID());
aGroupingItemsUsageCtl2.setObjectType(new Double(2));

Vector vRelaitonship = new Vector();
vRelaitonship.addElement(aGroupingItemsUsageCtl1);
vRelaitonship.addElement(aOrganizationCtl);

Vector vRelaitonship2 = new Vector();
```

FIG. 10a

10/76

```
vRelaitonship2.addElement(aGroupingItemsUsageCtl2);
vRelaitonship2.addElement(aFacilityCtl);

aFacilityCtl.addQuery(vRelaitonship);
aFacilityCtl.addQuery(vRelaitonship2);

Vector vaFacilityCtl=new Vector();
ii =aFacilityCtl.LoadSet(vaFacilityCtl);
```

FIG 10b

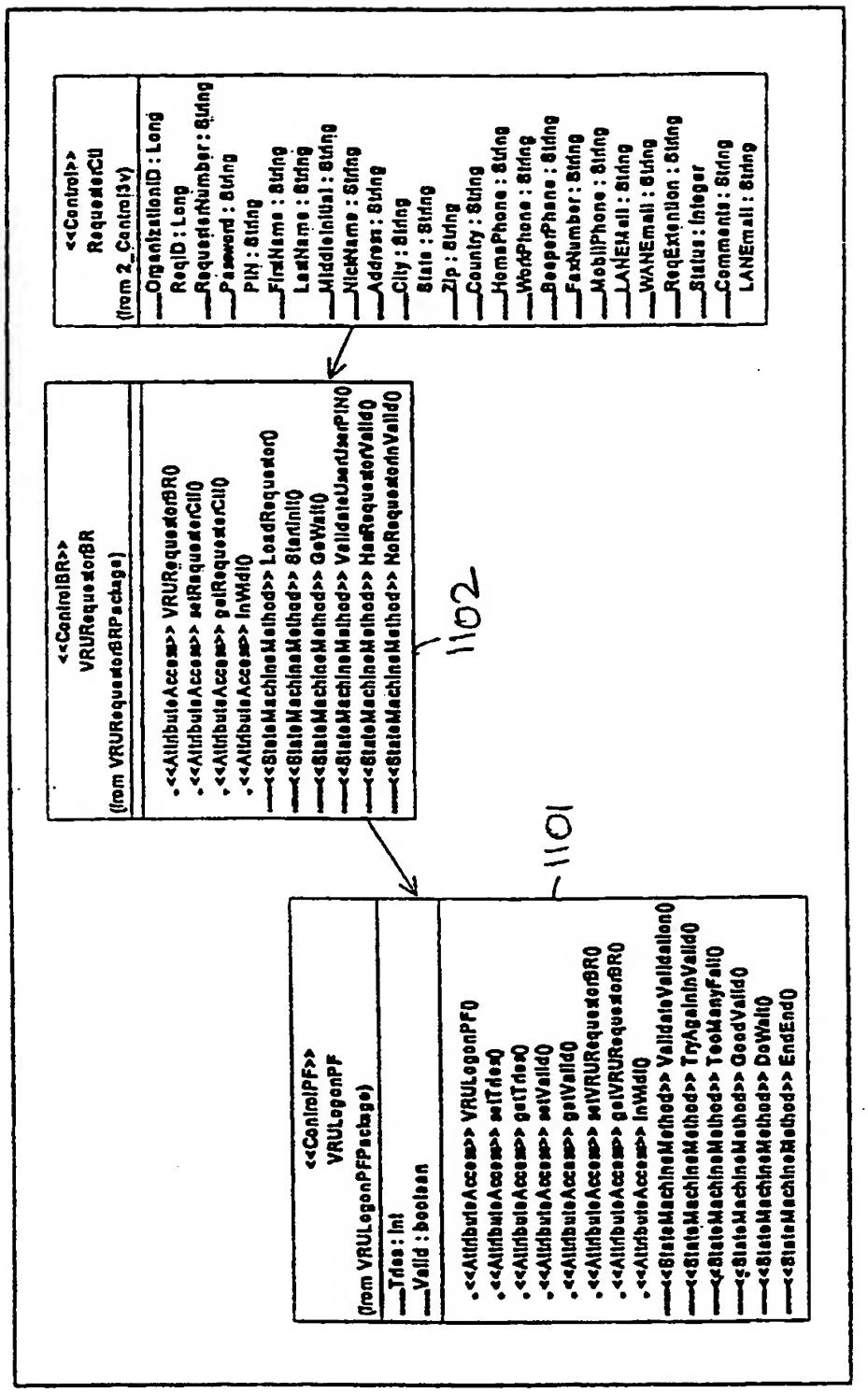

<<Control>>
RequesterCtl
(from 2_Control3v)
OrganizationID : Long
ReqID : Long
RequesterNumber : String
Password : String
PIN : String
FirstName : String
LastName : String
MiddleInitial : String
NickName : String
Address : String
City : String
State : String
Zip : String
County : String
HomePhone : String
WorkPhone : String
BeeperPhone : String
FaxNumber : String
MobilPhone : String
LANEMail : String
WANEmail : String
ReqExtention : String
Status : Integer
Comments : String
LANEmail : String
<<ControlBR>>
VRURequestorBR
(from VRURequestorBRPackage)
<<AttributeAccess>> VRURequestorBR()
<<AttributeAccess>> setRequesterCtl()
<<AttributeAccess>> getRequesterCtl()
<<AttributeAccess>> InWdl()
<<StateMachineMethod>> LoadRequestor()
<<StateMachineMethod>> StartInit()
<<StateMachineMethod>> GetWait()
<<StateMachineMethod>> ValidateUserUserPIN()
<<StateMachineMethod>> HasRequestorValid()
<<StateMachineMethod>> NoRequestorInValid()
1102
<<ControlPF>>
VRULogonPF
(from VRULogonPFPackage)
Tries : Int
Valid : boolean
<<AttributeAccess>> VRULogonPF()
<<AttributeAccess>> setTries()
<<AttributeAccess>> getTries()
<<AttributeAccess>> setValid()
<<AttributeAccess>> getValid()
<<AttributeAccess>> setVRURequestorBR()
<<AttributeAccess>> getVRURequestorBR()
<<AttributeAccess>> InWdl()
<<StateMachineMethod>> ValidateValidation()
<<StateMachineMethod>> TryAgainInValid()
<<StateMachineMethod>> TooManyFail()
<<StateMachineMethod>> GoodValid()
<<StateMachineMethod>> DoWait()
<<StateMachineMethod>> EndEnd()
1101

FIG. 11

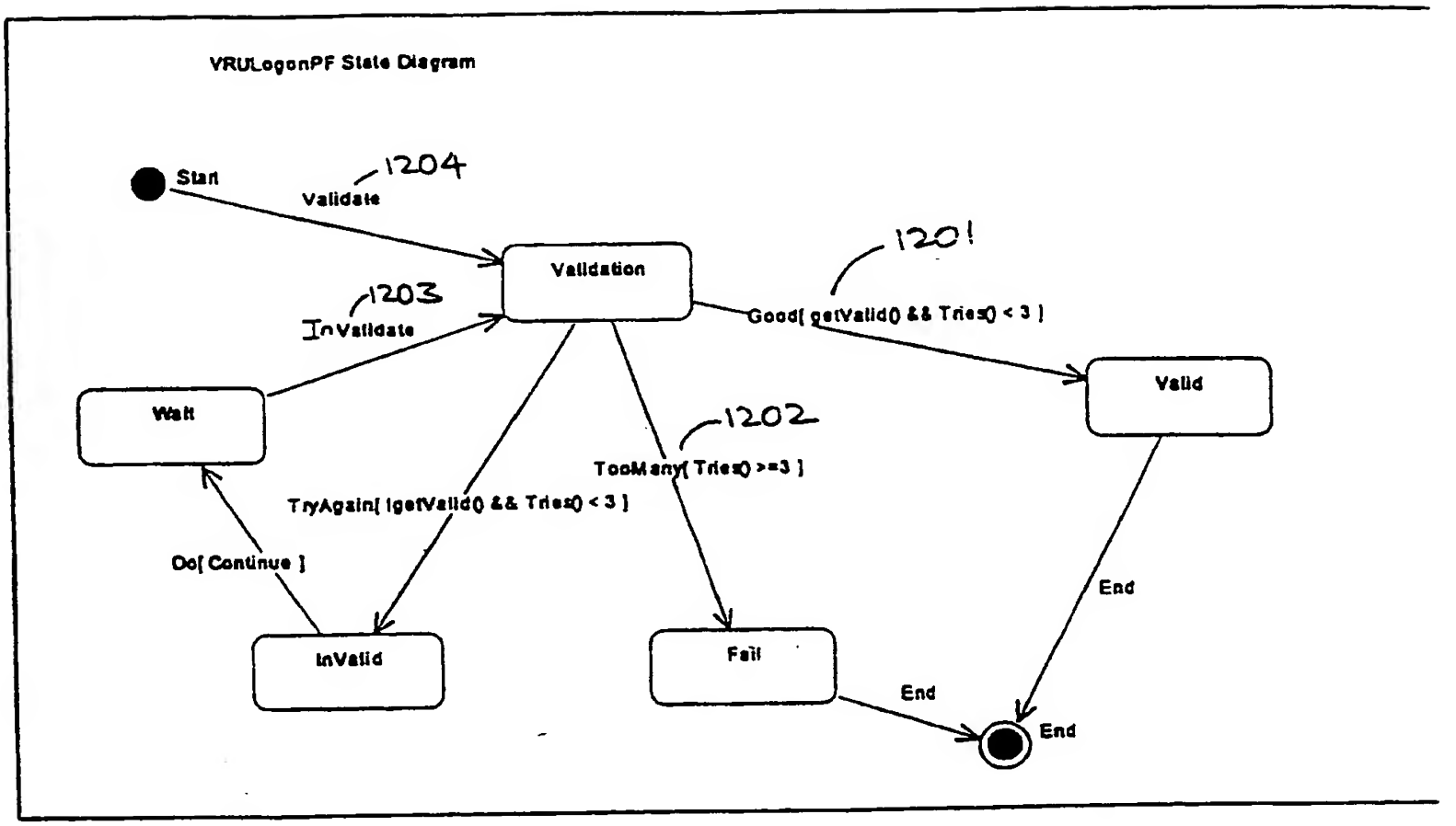

VRULogonPF State Diagram
Start
Validate
1204
Validation
1201
InValidate
1203
Good[getValid() && Tries() < 3]
Valid
Wait
1202
TooMany[Tries() >=3]
TryAgain[!getValid() && Tries() < 3]
Do[Continue]
InValid
Fail
End
End
End

FIG. 12

```
/*****TWG Rational Generation*****
                    // Generated StateMachine

                       @roseuid -3  9
                       Regenerate Code: Yes
                    *****TWG Rational Generation*****/
  public void resetStateMachine(){
     mCurrentState="Start";
  }

  public boolean inWidl(Widl theWidl){
        boolean synchronous = false;
        boolean Continue = true; // used to let the rational model read
      well on pass through states.
        boolean ret = true;
        int iState = -1;
        int iEvent = -1;
        String method ="";
```

FIG. 13a

```
    do{
       method = theWidl.getMethodName();
       try {
         iState =
((Integer)mStateHash.get(mCurrentState)).intValue();
       } catch (Exception e){iState=-1;}
       try {
         iEvent = ((Integer)mEventHash.get(method)).intValue();
       } catch (Exception e){iState=-1;}
       switch(iState){
          case isStart:
             switch (iEvent){
                case ieValidate:
                   mCurrentState = sValidation;
                   ValidateValidation( theWidl);
                   if (!getValid() && Tries() < 3){
                      theWidl.setMethodName( eTryAgain);
                      synchronous = true;
                   }
                   if (Tries() >=3){
                      theWidl.setMethodName( eTooMany);
                      synchronous = true;
                   }
                   if (getValid() && Tries() < 3){
                      theWidl.setMethodName( eGood);
                      synchronous = true;
                   }
                break;

             }; // end switch (iEvent){
          break; // isStart
          case isValidation:
             switch (iEvent){
                case ieTryAgain:
                   mCurrentState = sInValid;
                   TryAgainInValid( theWidl);
                   if (Continue){
                      theWidl.setMethodName( eDo);
                      synchronous = true;
                   }
                break;

                case ieTooMany:
                   mCurrentState = sFail;
                   TooManyFail( theWidl);
                   synchronous = false;
                   return true;

                case ieGood:
                   mCurrentState = sValid;
                   GoodValid( theWidl);
                   synchronous = false;
                   return true;

             }; // end switch (iEvent){
          break; // isValidation
          case isInValid:
             switch (iEvent){
                case ieDo:
                   mCurrentState = sWait;
                   DoWait( theWidl);
                   synchronous = false;
```

FIG. 13b

```
            return true;

         }; // end switch (tEvent){
      break; // isInValid
      case isWait:
         switch (iEvent){
            case ieValidate:
               mCurrentState = sValidation;
               ValidateValidation( theWidl);
               if (!getValid() && Tries() < 3){
                  theWidl.setMethodName( eTryAgain);
                  synchronous = true;
               }
               if (Tries() >=3){
                  theWidl.setMethodName( eTooMany);
                  synchronous = true;
               }
               if (getValid() && Tries() < 3){
                  theWidl.setMethodName( eGood);
                  synchronous = true;
               }
            break;

         }; // end switch (iEvent){
      break; // isWait
      case isFail:
         switch (iEvent){
            case ieEnd:
               mCurrentState = sEnd;
               EndEnd( theWidl);
               synchronous = false;
            break;

         }; // end switch (iEvent){
      break; // isFail
      case isValid:
         switch (iEvent){
            case ieEnd:
               mCurrentState = sEnd;
               EndEnd( theWidl);
               synchronous = false;
            break;

         }; // end switch (iEvent){
      break; // isValid
   }; // end of Switch(iState)
} while (synchronous == true);
return ret;
}

            /*****TWG End @roseuid -3 ***/
```

FIG 13C

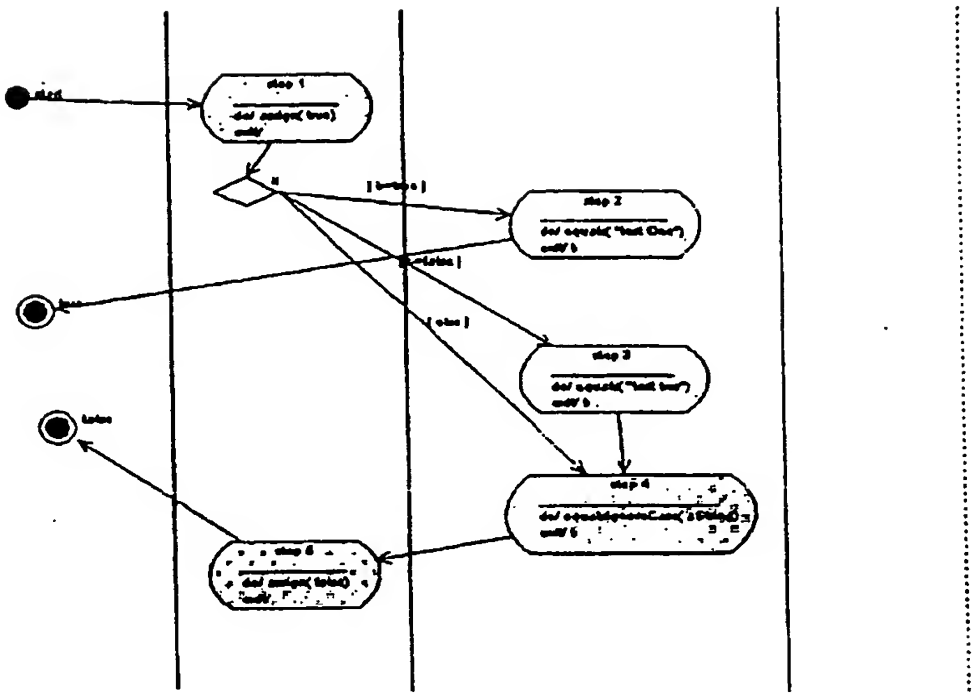


FIG. 14

```
                    /*****TWG Rational Generation*****
                    //  <P>
                    // StateMachine Documentation: <P>
                    // Place Model notes before StateMachine
Documentation
                    //  label!! <P>Event:  <P>State: Set Widl Return
result=Fail

                       @roseuid 371DD79F020B  9
                       Regenerate Code: Yes
                    *****TWG Rational Generation*****/
  protected void TooManyFail(Widl theWidl) {
   try{
                    /**
                    //  <P>
                    // reciever Widl theWidl:: sender VRULogonPF this
<P>
                    setReturnParameter("result", "Fail")
                    **/
          theWidl.setReturnParameter("result", "Fail");
     } catch (Exception e){
       System.err.println("VRULogonPF.TooManyFail: " + e);
       e.printStackTrace();
     }
   }

                    /*****TWG End @roseuid 371DD79F020B ***/
```

FIG. 15

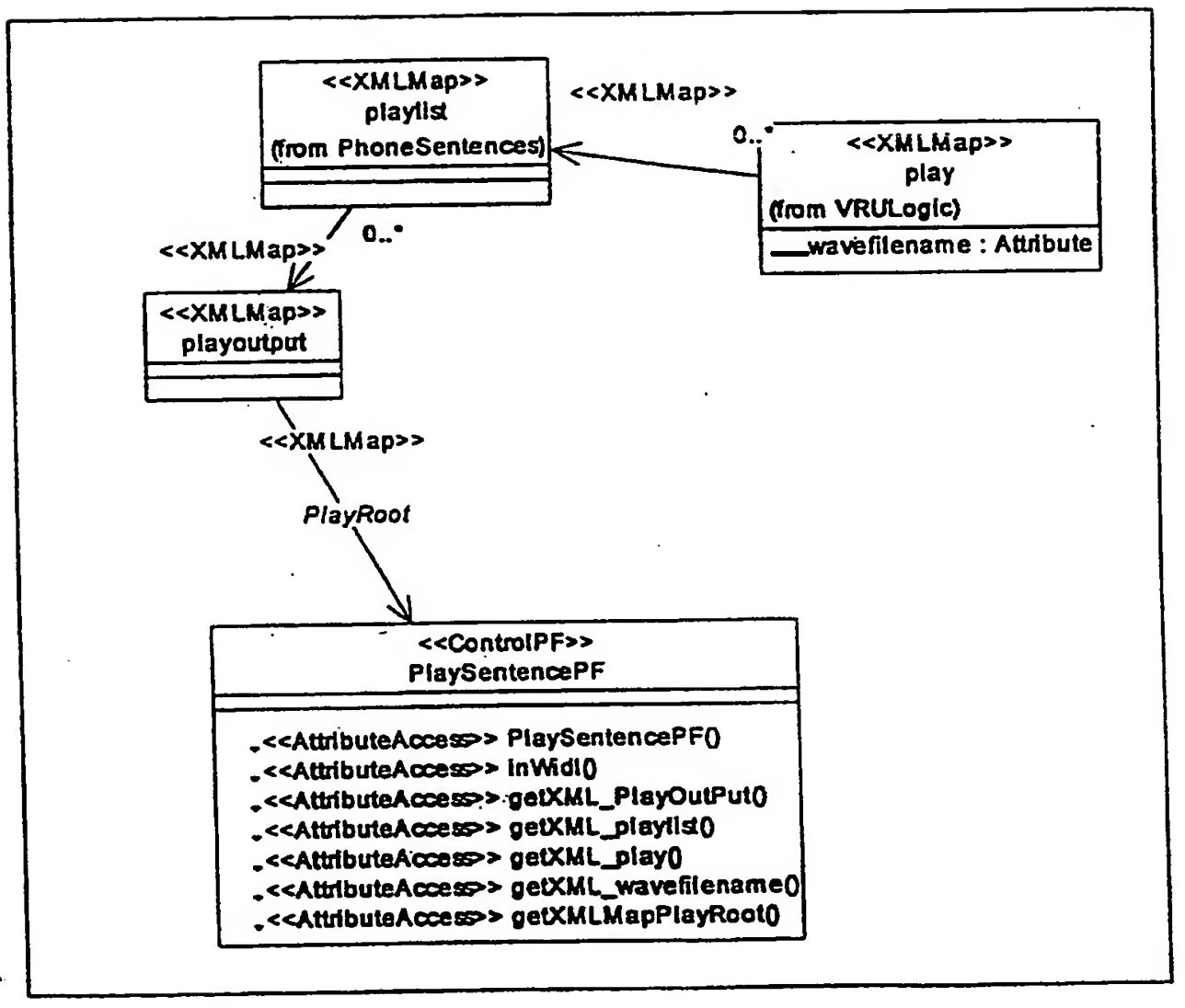

<<XMLMap>>
playlist
(from PhoneSentences)
<<XMLMap>>
0..*
<<XMLMap>>
play
(from VRULogic)
__wavefilename : Attribute
<<XMLMap>>
0..*
<<XMLMap>>
playoutput
<<XMLMap>>
PlayRoot
<<ControlPF>>
PlaySentencePF
<<AttributeAccess>> PlaySentencePF()
<<AttributeAccess>> InWid()
<<AttributeAccess>> getXML_PlayOutPut()
<<AttributeAccess>> getXML_playlist()
<<AttributeAccess>> getXML_play()
<<AttributeAccess>> getXML_wavefilename()
<<AttributeAccess>> getXMLMapPlayRoot()

FIG. 16

```
/*****TWG Rational Generation*****

                @roseuid -4  10
                Regenerate Code: Yes
                *****TWG Rational Generation*****/

        static {
            mXMLMapPlayRoot = new XMLMap();
            mXMLMapPlayRoot.addMapping("PlayOutPut", "xml");
            mXMLMapPlayRoot.addMapping("playlist", "xml* /playlist");
            mXMLMapPlayRoot.addMapping("play", "xml/playlist* /play");
            mXMLMapPlayRoot.addMapping("wavefilename",
            "xml/playlist/play#wavefilename");

        } // end static
```

FIG. 17

```
<!ELEMENT xml playlist>
<!ELEMENT playlist, play>
<!ELEMENT play>
    <!ATTLIST wavefilename VALUE CDATA >
```

FIG 18

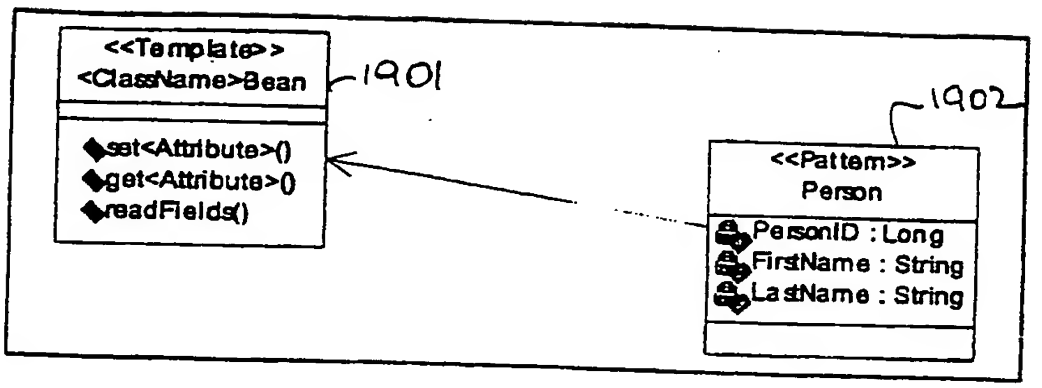

<<Template>>
<ClassName>Bean
1901
set<Attribute>()
get<Attribute>()
readFields()
1902
<<Pattern>>
Person
PersonID : Long
FirstName : String
LastName : String

FIGURE 19

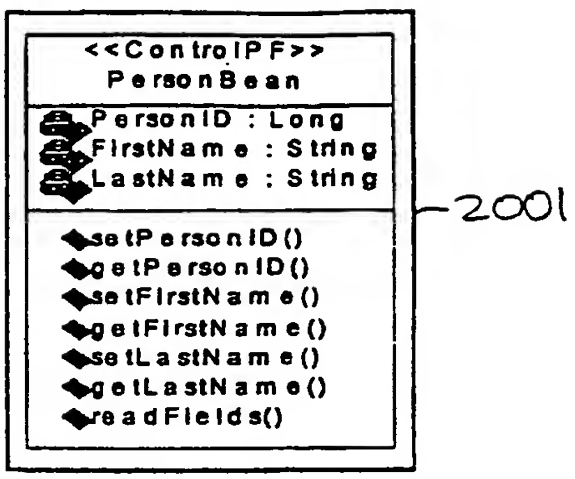

<<ControlPF>>
PersonBean
PersonID : Long
FirstName : String
LastName : String
setPersonID()
getPersonID()
setFirstName()
getFirstName()
setLastName()
getLastName()
readFields()
2001

FIGURE 20

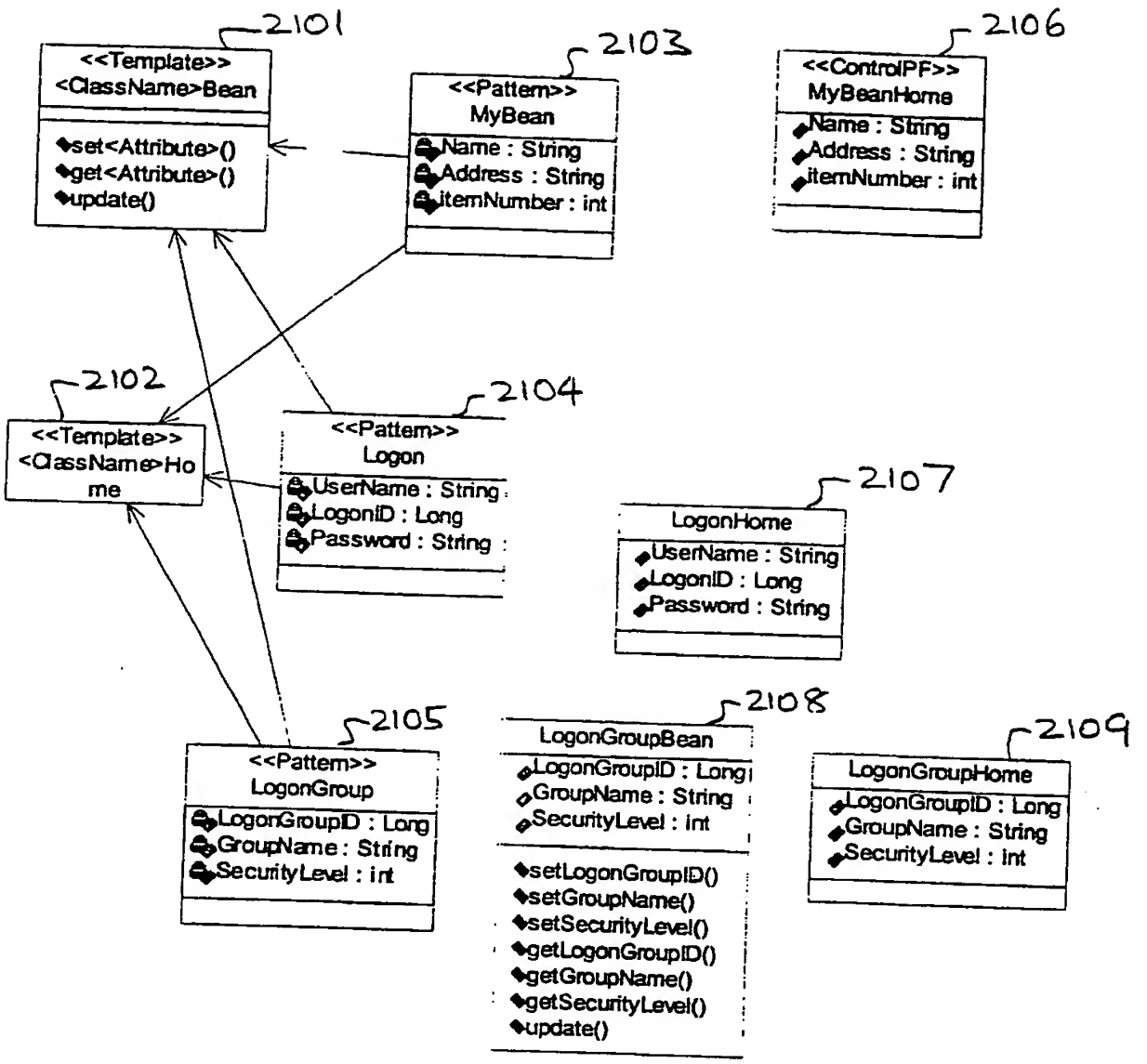

2101
<<Template>>
<ClassName>Bean
set<Attribute>()
get<Attribute>()
update()
2103
<<Pattern>>
MyBean
Name : String
Address : String
itemNumber : int
2106
<<ControlPF>>
MyBeanHome
Name : String
Address : String
itemNumber : int
2102
<<Template>>
<ClassName>Ho
me
2104
<<Pattern>>
Logon
UserName : String
LogonID : Long
Password : String
2107
LogonHome
UserName : String
LogonID : Long
Password : String
2105
<<Pattern>>
LogonGroup
LogonGroupID : Long
GroupName : String
SecurityLevel : int
2108
LogonGroupBean
LogonGroupID : Long
GroupName : String
SecurityLevel : int
setLogonGroupID()
setGroupName()
setSecurityLevel()
getLogonGroupID()
getGroupName()
getSecurityLevel()
update()
2109
LogonGroupHome
LogonGroupID : Long
GroupName : String
SecurityLevel : int

FIGURE 21

Introduction

- **Effective management and modeling of enterprise applications**
- **Web and business-to-business applications**
- **Messaging environments**
  - loosely coupled
  - asynchronous
  - fault tolerant
- **Java technology provides the descriptive language**
- **XML provides the data representation.**
- **UML provides the notational language.**
- **Manage complexity for deploying n-tier enterprise applications**

FIG. 22

WO 01/08007
PCT/US00/20069
25/76

Overview

- **A process and examples for building UML applications with XML messages through multiple distributed server containers.**
- **Illustrate a complete UML design for N-Tier Application Web Development**
- **Implement a web based logon and user profile application.**
- **Utilizing**
  - UML
  - Java technology
  - XML DTD/Schema definitions

FIG. 23

Tools

- Examples presented in UML using Rational Rose® with UML Factory®.
- The examples will be implemented with JAR components that can be deployed into multiple configurations.
- UML Factory will generate, deploy, and animate the examples through the UML diagrams.

FIG. 24

WO 01/08007
PCT/US00/20069
27/76

Technologies

- **UML:**
  - Unified Modeling Language will describe the static and dynamic behavior for applications.
- **XML:**
  - Extensible Markup Language describes document information for building pages and carrying messages through the application tiers.
- **JSP™ components:**
  - JavaServer Pages™ components are used to define dynamic HTML interface pages

FIG. 25

Technologies

- **XSL:**
  - Extensible Style Sheet language is used to transform the XML documents into dynamic HTML interface pages.
- **EJB™**
  - Enterprise JavaBeans™ specification is used for data access and manipulation.
- **JMS:**
  - Java™ Message Service API provides an asynchronous fault tolerant messaging capability between application tiers.

FIG. 26

WO 01/08007
PCT/US00/20069
29/76

Architectural Overview

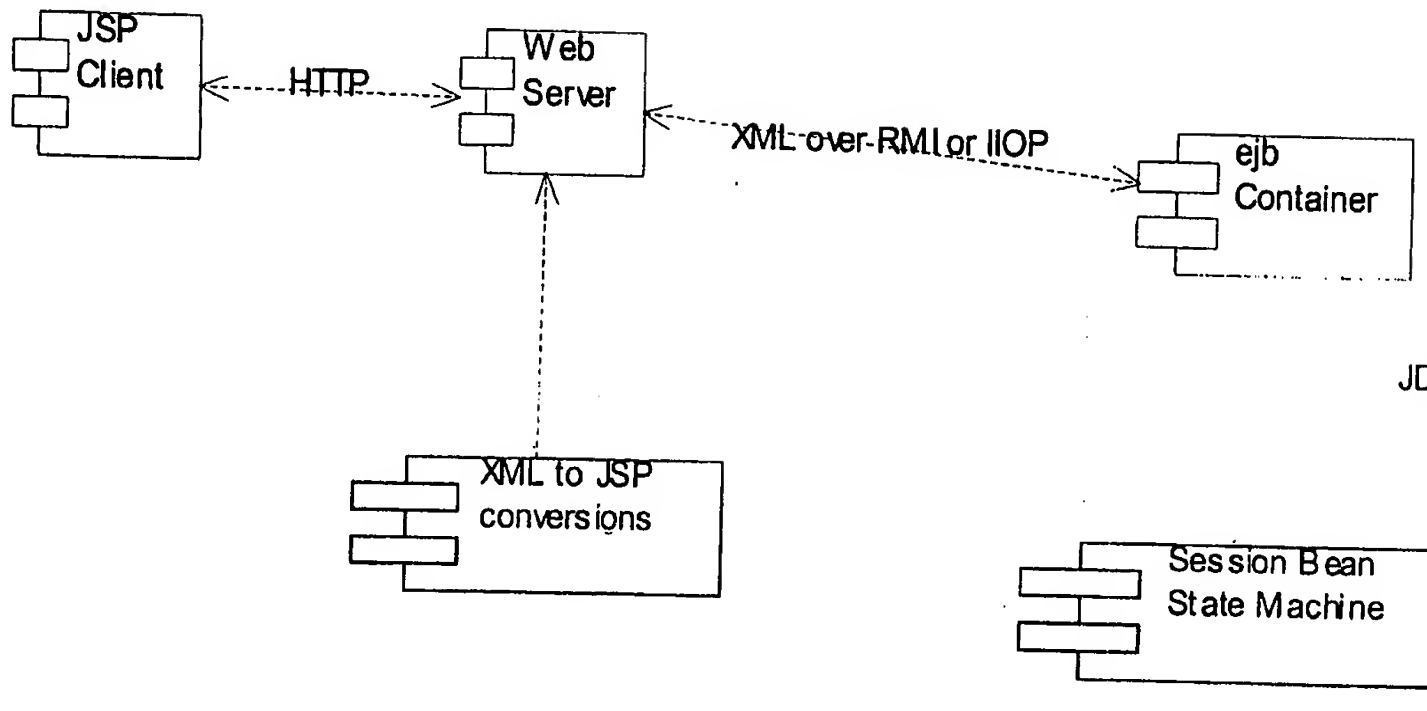


FIG. 27

UML Model Overview

- Use Case Diagram
- Collaboration Diagram
- Class Diagrams
- State Diagrams
- Activity Diagrams
- Deployment Diagram

FIG. 28

Modeling the Application

- **Use Case**
  - Use Case Diagrams define the high-level interactions between external actors and system processes. The Use Case diagram must have an actor, an interaction, and a process.

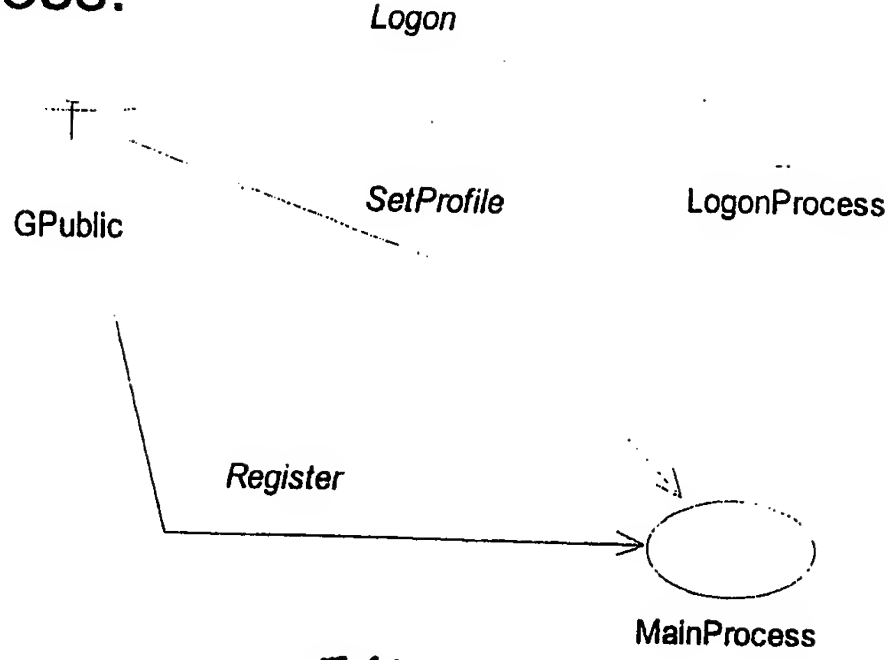


FIG. 29

WO 01/08007
PCT/US00/20069
32/76

Alternate Implementations

- **Realize.**
  - The “realize" stereotype on a use case interaction defines alternate mechanisms for implementing the same process.

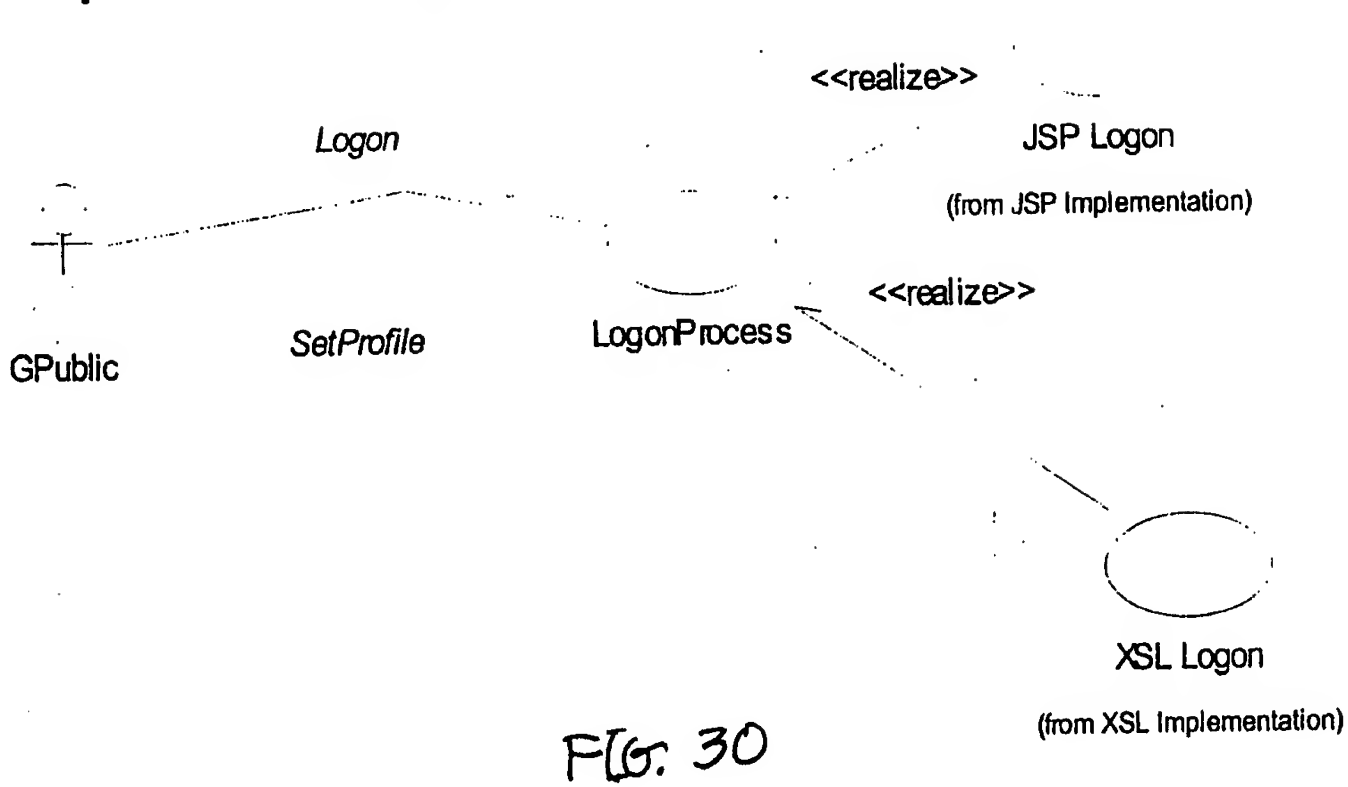


FIG. 30

WO 01/08007
PCT/US00/20069

User Artifacts

- Each interaction on a use case process contains artifacts, these artifacts are tangible attributes provided to the actor or input artifacts from the actor to the process. For a logon:
  - Username
  - UserPassword

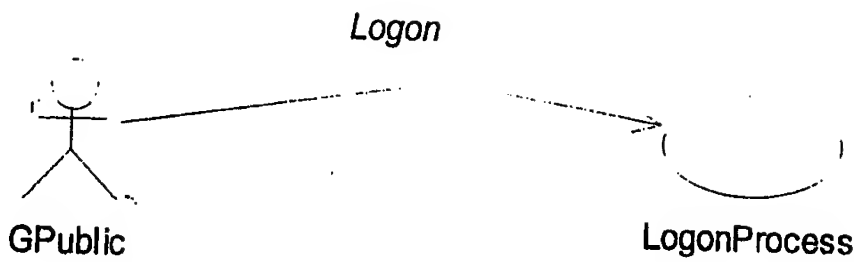


FIG. 31

Collaboration Diagram

– Collaboration diagrams define the implementation for each use case interaction.

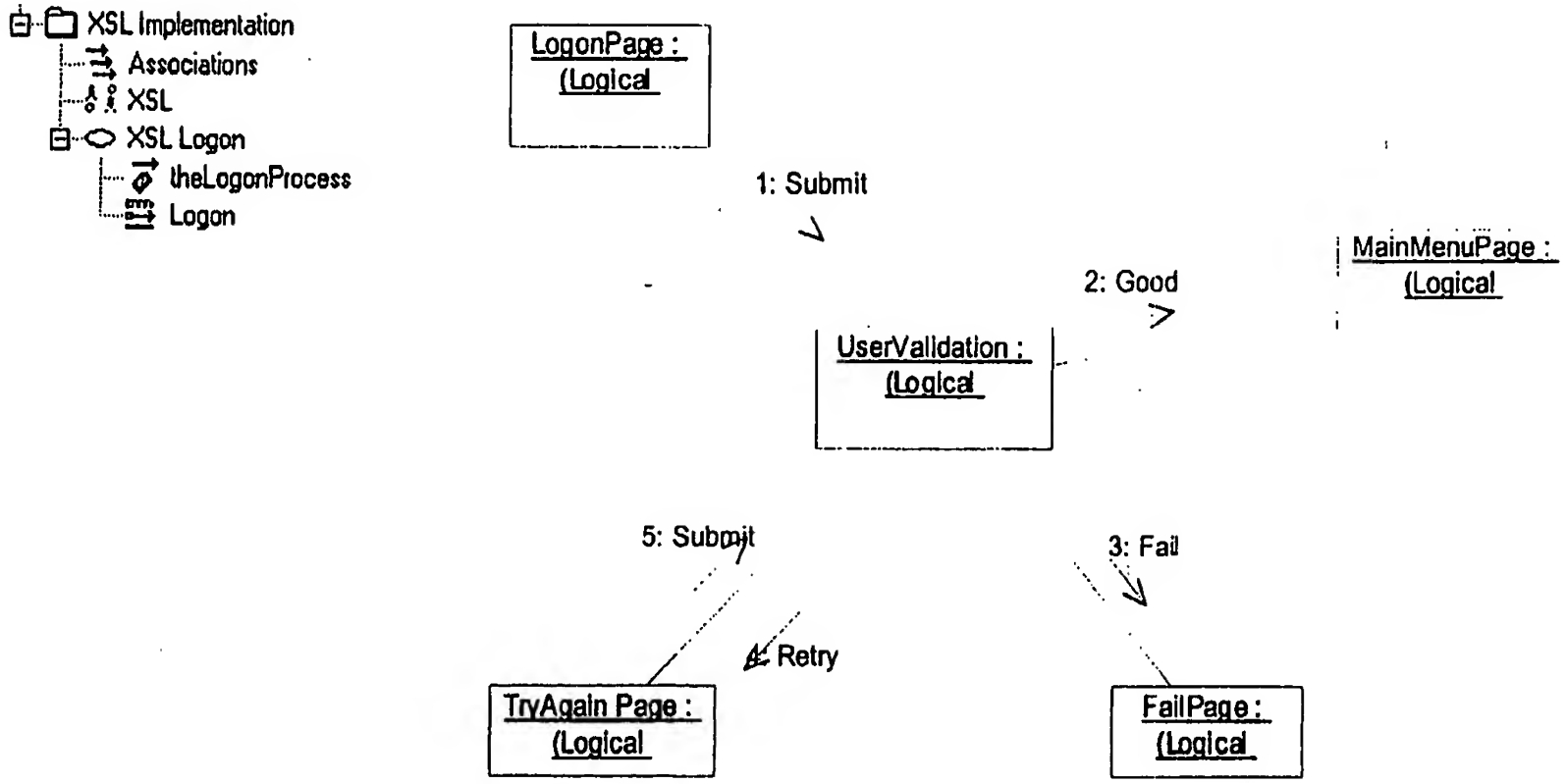
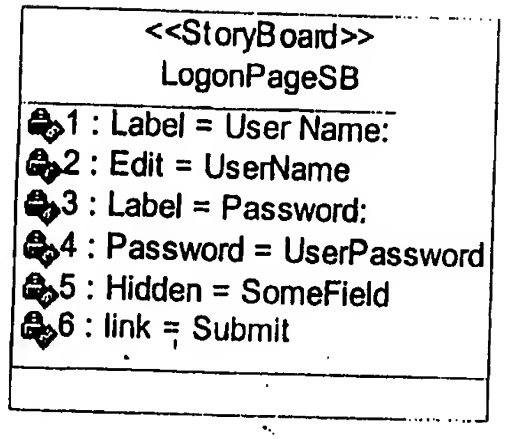


FIG. 3J

WO 01/08007
PCT/US00/20069
35/76

Storyboard Interface Objects

– Storyboard interface objects identify user input and output artifacts. These objects will identify the types of artifacts used within the system for the use case interaction.



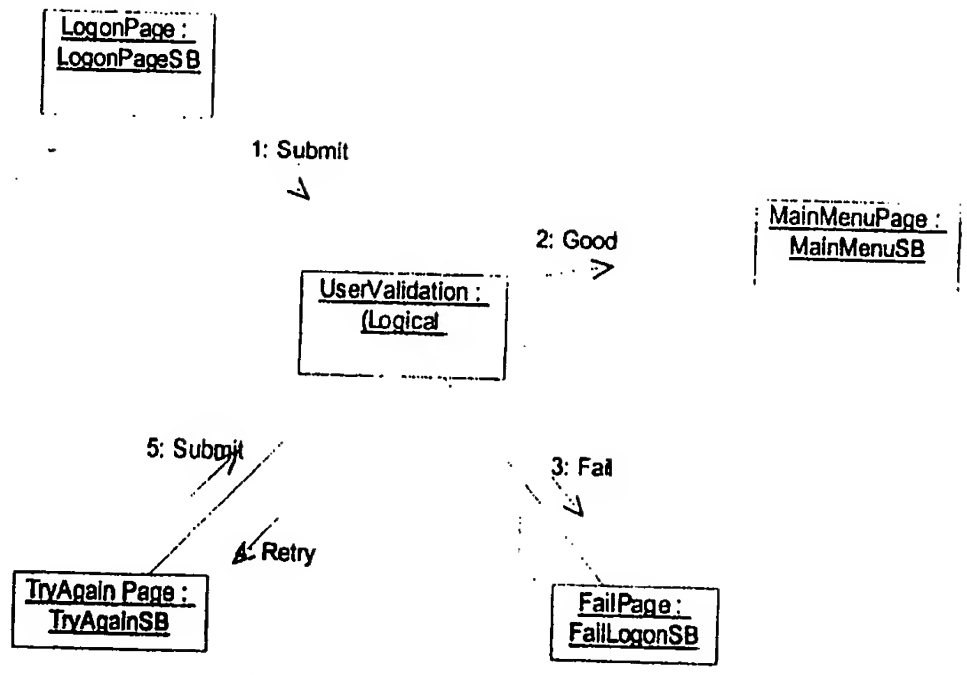


FIG. 33

WO 01/08007
PCT/US00/20069
36/76

Decision Control Process Flow Objects

– Through the collaboration diagram choices or decisions are made directing the diagram flow. Control process flow objects define the domain logic class specifications that will govern these decisions.

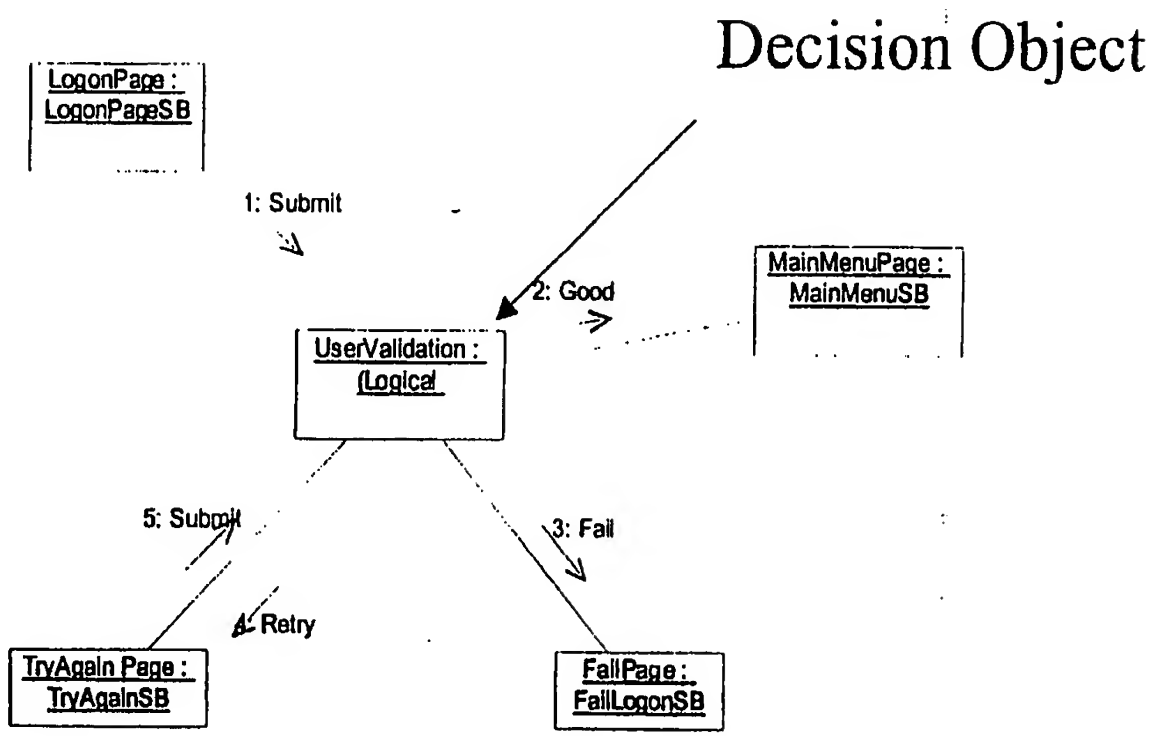


FIG. 34

Events

– Collaboration diagram events show the linking between various collaboration diagram objects.

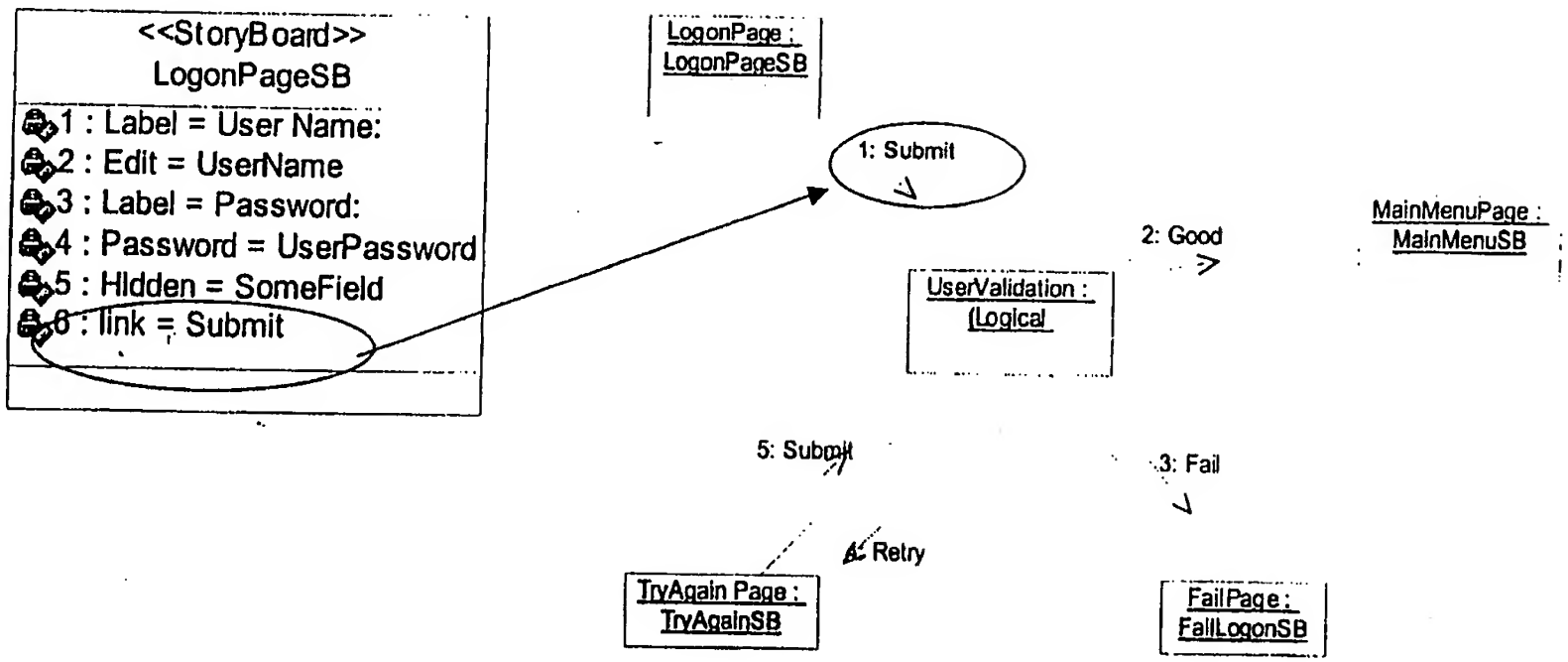


FIG. 35

Storyboard Class Specifications

– The storyboard class specification identifies the ordered set of artifacts and events that this storyboard object element will contain.

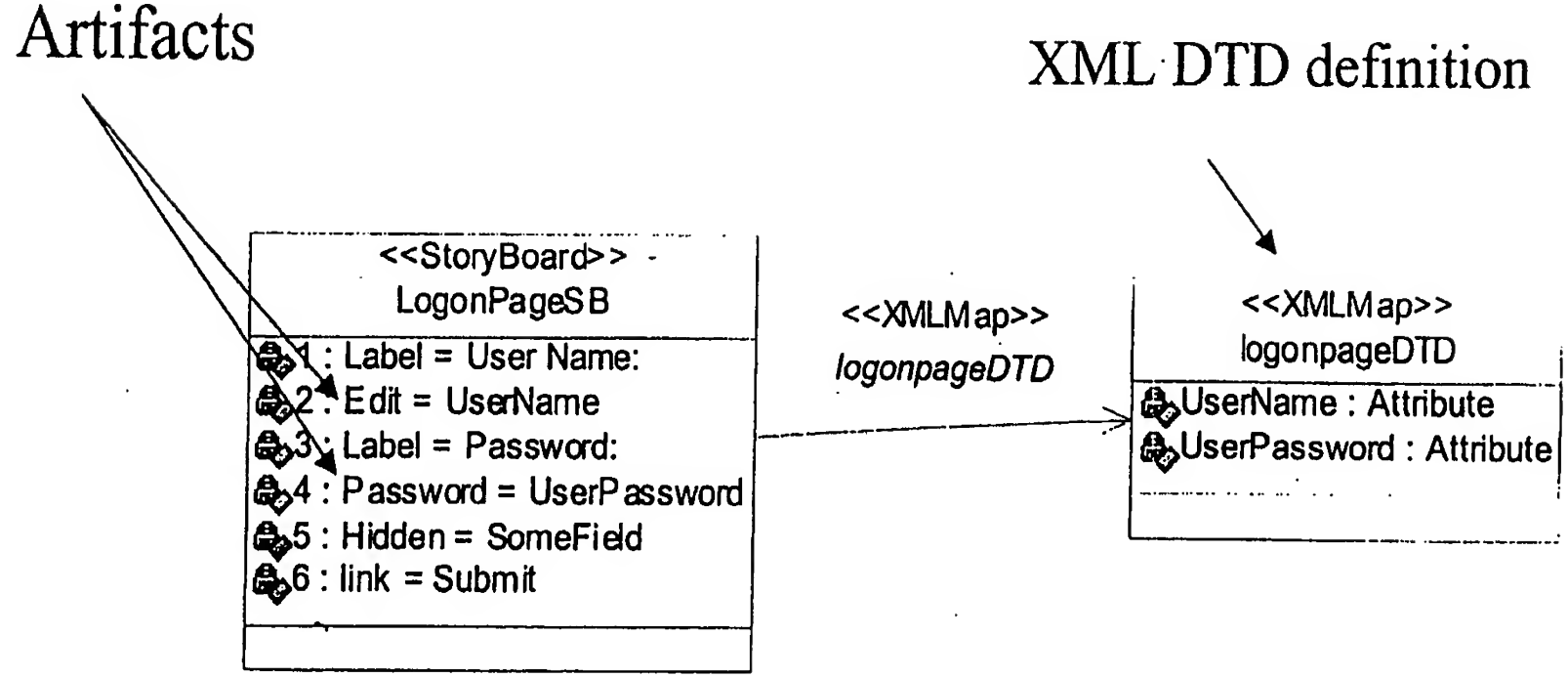


FIG. 36

WO 01/08007
PCT/US00/20069
39/76

HTML Pages

– Each storyboard class specification is linked to an HTML page that will be processed to create the XSL or JSP interface definition.

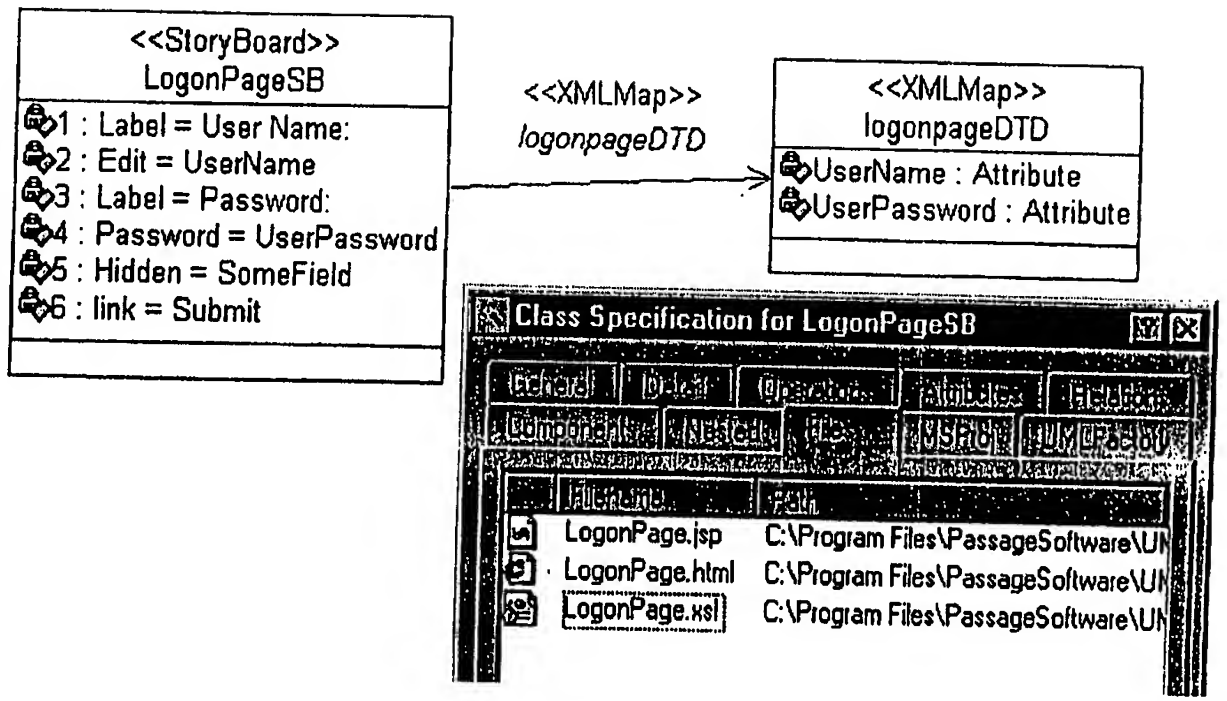


FIG. 37

HTML Pages

– The HTML pages will contain tokens identifying the XML abstract semantic names representing the use case artifacts coming and going from the control process flow.

FIG. 38

XML DTD/Schema Modeling

– Modeling the XML schema information within UML provides a visual representation of the XML documents structure. The modeled XML document also provides runtime information and model time checking for collating the use case artifacts with the XML elements.

• Sample Company Person XML

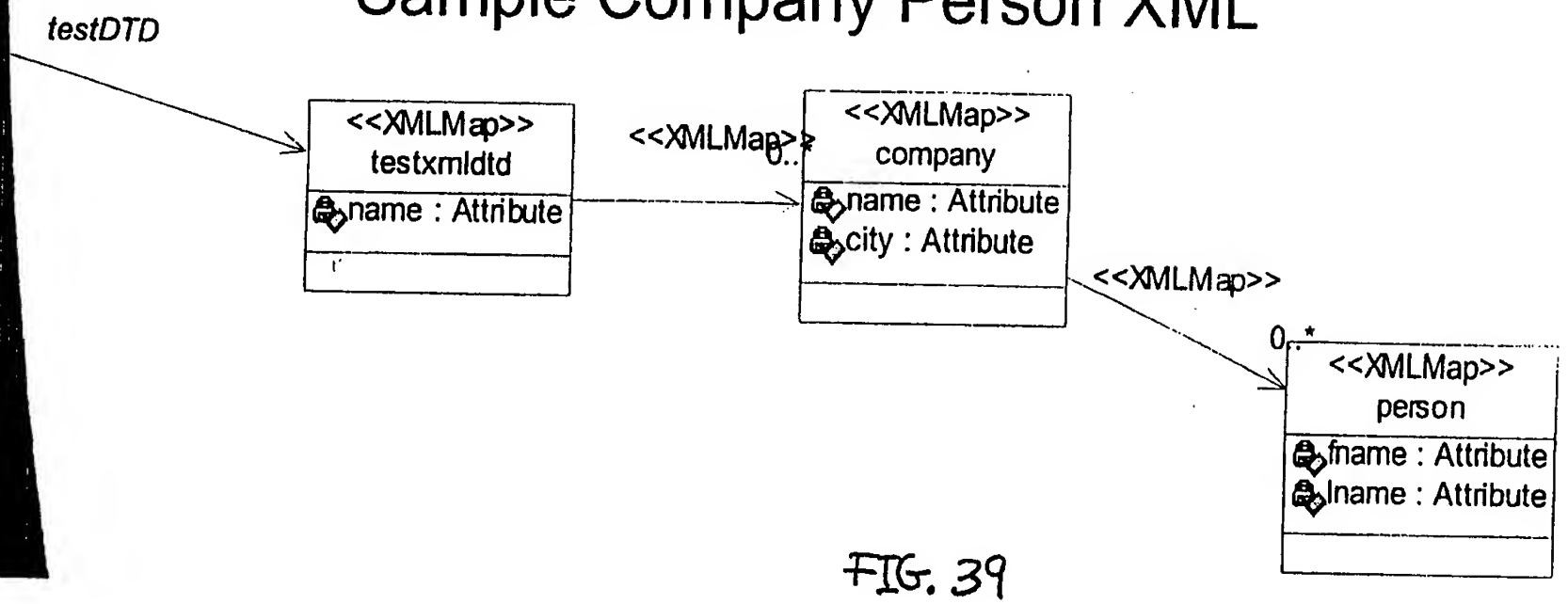


FIG. 39

XML Mappings

- **UML Factory Provides an abstract mapping between XML elements and semantic names. These Mappings allow isolation between the arbitrary physical representation of a data element, and a logical name or handle to access and manipulate that element.**
  - *Semantic Names*
    - Semantic Names Identify abstract textual identifiers for elements, or Attributes, within the XML document.
  - *XML Element Mappings*
    - XML element mappings identify the arbitrary physical XML element definition. The XML mappings identify XML text, cardinality, attributes, and schema information

FIG. 40

WO 01/08007
43/76
PCT/US00/20069

XML Mapping example

– Generated semantic mappings to sample Company Person XML

```
mXMLMaptestDTD = new XMLMap();
mXMLMaptestDTD.addMapping("TestXML", "xml");
mXMLMaptestDTD.addMapping("XmlName", "xml/name");
mXMLMaptestDTD.addMapping("Company", "xml /company");
mXMLMaptestDTD.addMapping("CompanyName", "xml/company#name");
mXMLMaptestDTD.addMapping("CompanyCity", "xml/company#city");
mXMLMaptestDTD.addMapping("Person", "xml/company* /person");
mXMLMaptestDTD.addMapping("FirstName", "xml/company/person#fname");
mXMLMaptestDTD.addMapping("LastName", "xml/company/person#lname");
```

FIG. 41

XML Mappings to "JSP tags"

- The token semantic names within the HTML page associate with storyboard class specifications are substituted with methods to extract semantic data from the XML document representing the Storyboard interface object.

FIG. 42

XML Mappings to XSL tags

– In like manner, the token semantic names within the HTML page associated with storyboard class specifications are replaced with XSL syntax to transform the XML document representing the Storyboard interface object.

FIG. 43

Storyboard Events

- **The storyboard class specification objects contain events that will fire into the application, transitioning through the designed collaboration diagrams.**
  - If the target object of a collaboration diagram event is another interface element then that storyboard interface element will be displayed.
  - If the target object is a decision point then the class specification defined for that decision logic would be sent the event through the UML state machine implementation.

FIG. 44

Storyboard Events

- **The Logon Page Submit link event.**

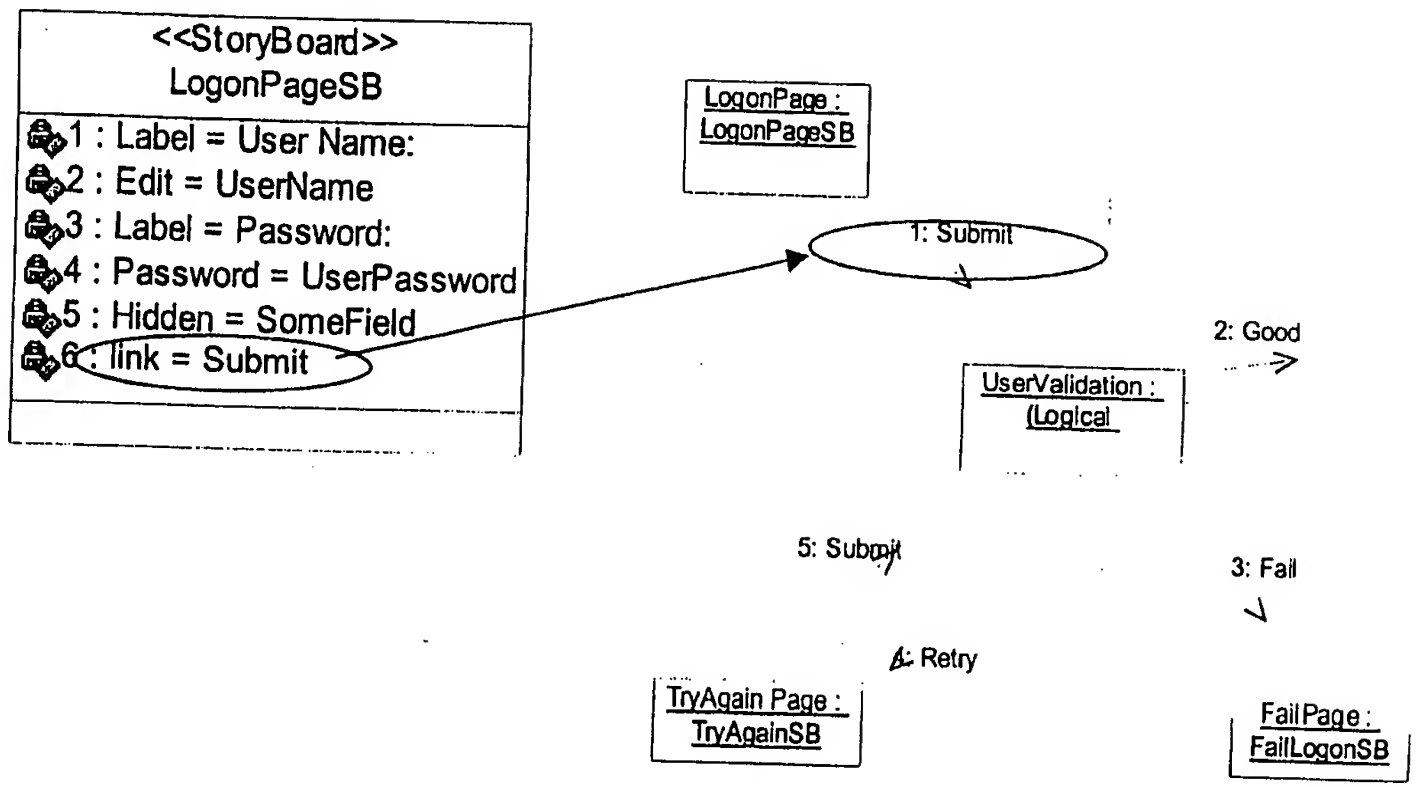


FIG. 45

Collaboration Decisions

– Decisions are implemented by UML logical classes

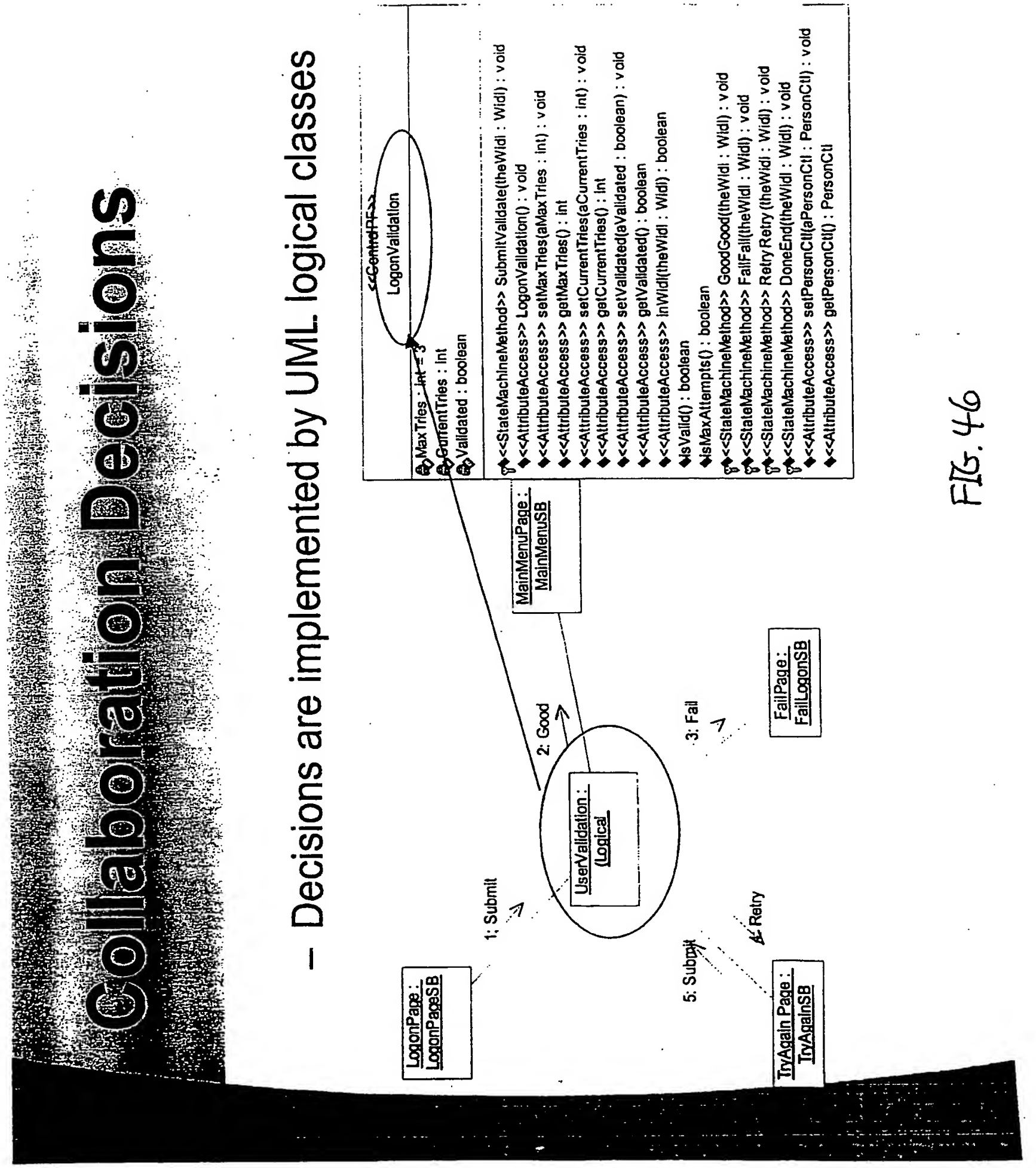


FIG. 46

Control Process Flow Class Specification

– The control process flow class specification provides the logical implementation making decisions through the use case collaboration process. The UML control process flow stereotyped class specification is generated into Java™ source code with all the static and dynamic behavior required for:

- Receiving input events
- Managing persistent data
- Manipulating Interface XML documents
- Returning information to the process.
- Maintaining state information of the application.

FIG. 47

WO 01/08007
50/76
PCT/US00/20069

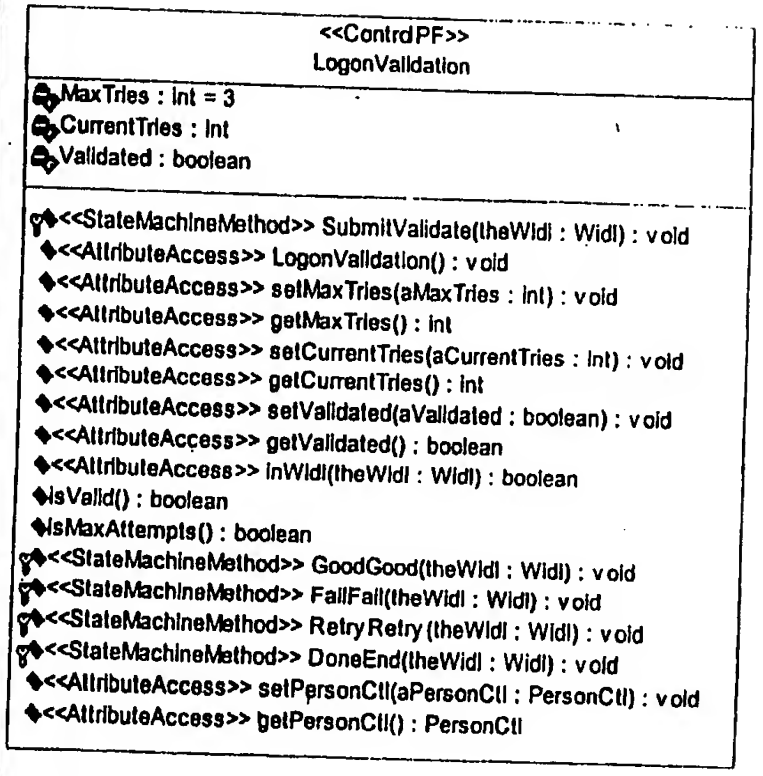
Receiving Events

– When a user activates an event from an interface object, that event is sent into the state machine. All information coming into the state machine is contained within an XML document.

FIG. 48

WO 01/08007
PCT/US00/20069
51/76

Control Process Class State Machines



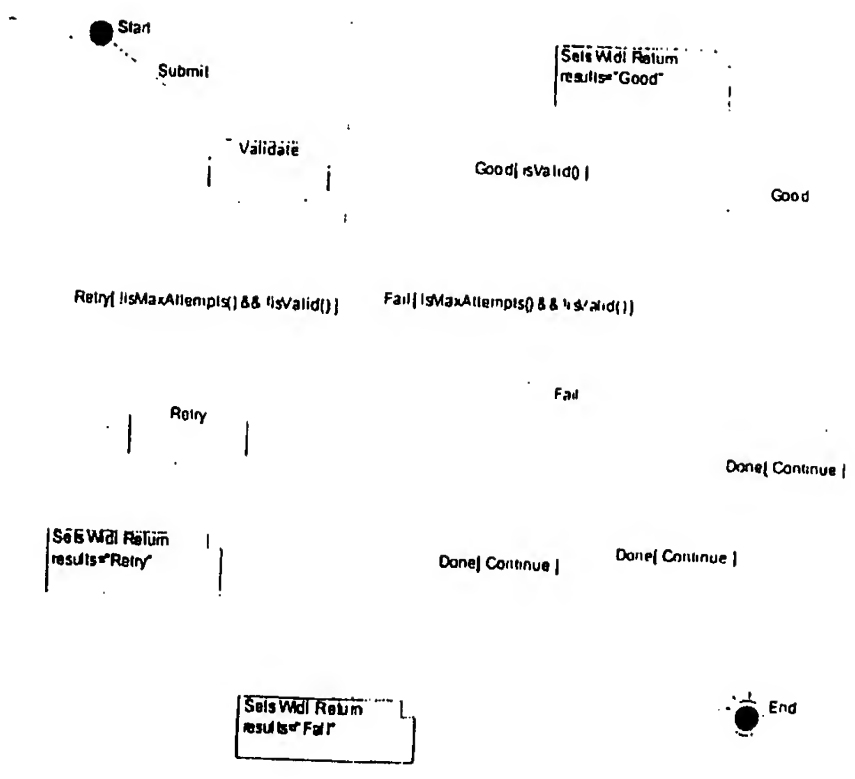


FIG. 49

WO 01/08007
PCT/US00/20069
52/76

WIDL

- **WIDL is a Web Interface Definition Language specification.**
  - The Widl contains an event, Process, and structured records defining behavior and data together within a single XML document package.
  - Event or Method
    - The method described within the Widl is the event name coming into the state machine.
  - Process
    - The Widl process attribute identifies the executable Java class that will receive the Widl and respond to the method event. The process name can be a fully qualified Java class or an abstract object name. Containers receiving the Widl input from either session beans, JMS messaging Queue implementations etc., must have a mechanism to late bind the Widl event to the correct process.

FIG. 50

WO 01/08007
PCT/US00/20069
53/76

WIDL Records

The Widl contains three records for input information, output information and return information. These records can be populated with attributes or complete XML documents.

- Input Record
  - Our example for the Web application will use the input record to contain information coming from the Web interface into the process logic.
- Output Record
  - The example Widl use the output record of the Widl to contain Return Page or XML document information back to the Web interface
- Return Record
  - The Widl return record contains results information for evaluating decisions to the collaboration process flow. The result Attribute within the Widl Return record contains the return events from the control process flow class specification.

FIG. 51

WO 01/08007
PCT/US00/20069
54/76

State Machines

– State machines define the possible events coming into a dynamic class and the state transitions for those events. The state machine diagram provides a readable definition of the dynamic behavior for a given class specification.

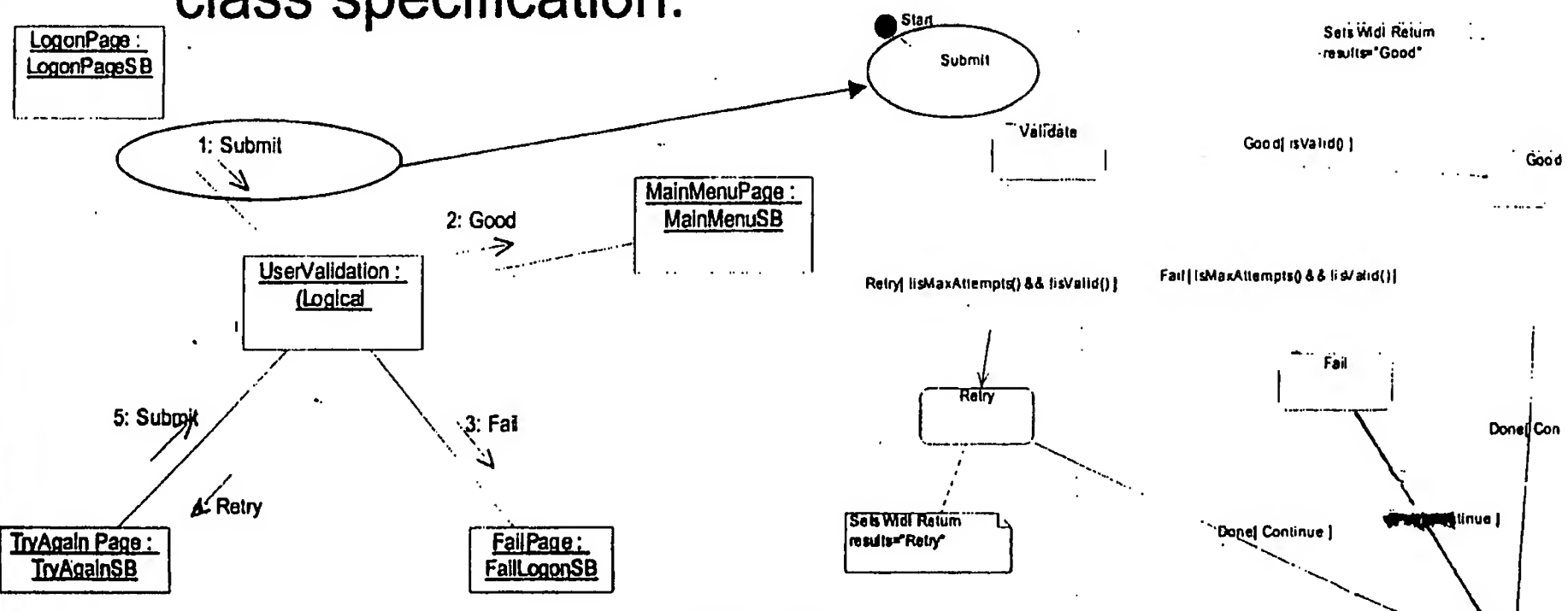


FIG 52

WO 01/08007
55/76
PCT/US00/20069

States and Transitions

- ***States***
  - States define the process stops through the state machine.
- ***Event Transitions***
  - Transitions define the event causing a transition from one state to another state.
- ***Event State Methods***
  - Event state methods are the implied action behaviors when an event transition occurs. These event state methods are implemented through UML activity diagrams.

FIG. 53

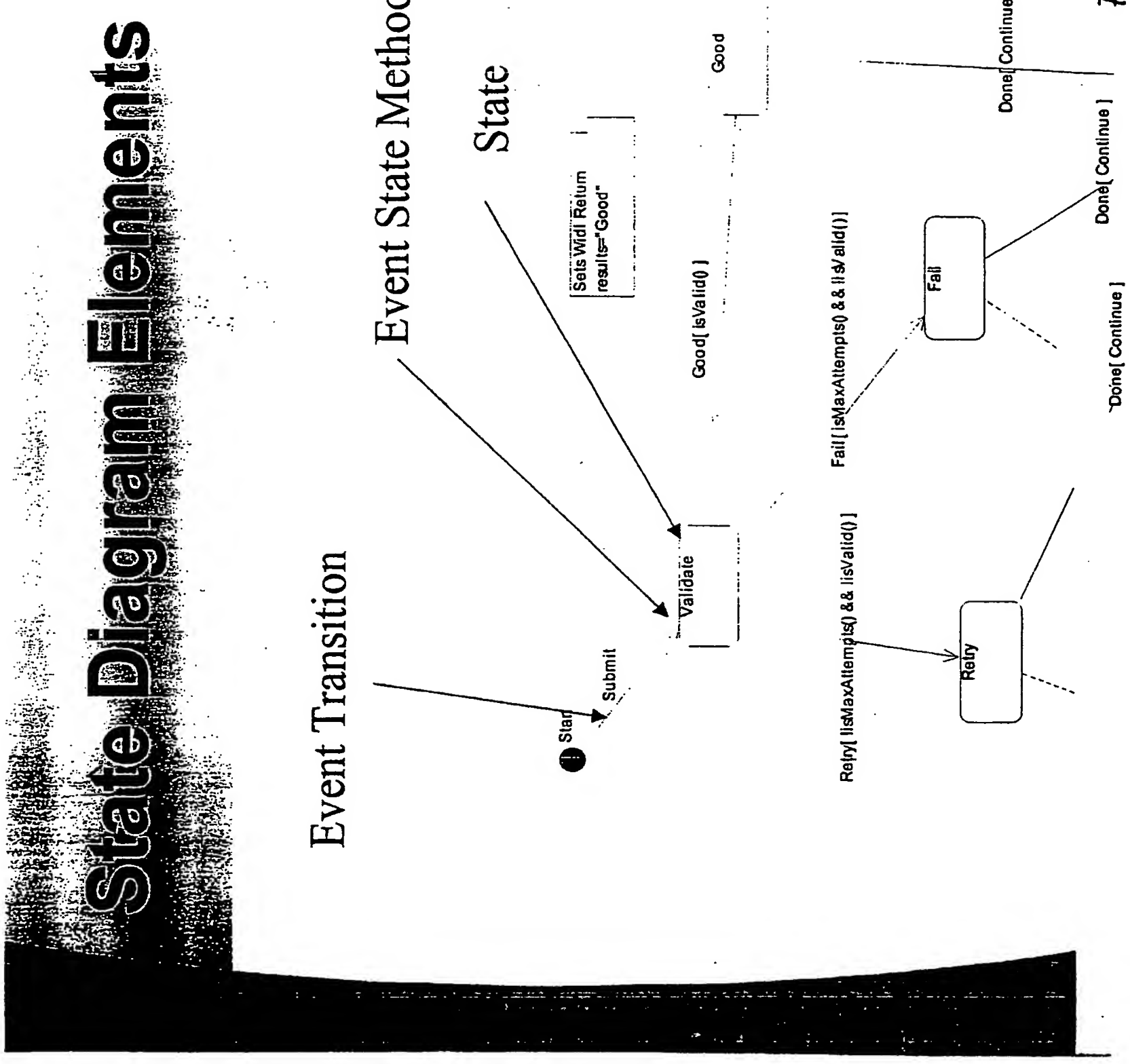

State Diagram Elements
Event State Method
State
Sets Widl Return result="Good"
Good[isValid()]
Good
Event Transition
Start
Submit
Validate
Fail[isMaxAttempts() && !isValid()]
Fail
Done[Continue]
Done[Continue]
Done[Continue]
Retry[!isMaxAttempts() && !isValid()]
Retry

FIG. 54

State Timing

- ***Guard Conditions***
  - Guard conditions imply a synchronous transition exiting a given state. The guard conditions contain boolean logic to determine which exit transition will be traversed.
- ***Asynchronous***
  - Asynchronous events have no guard conditions and are triggered from some external source. That source is typically a user interface link artifacts.
- ***Synchronous***
  - Synchronous events transition automatically and internal to the state machine. Synchronous events are identified as exit events from a state containing guard conditions.

FIG. 55

WO 01/08007
PCT/US00/20069
58/76

Transition Timing Example

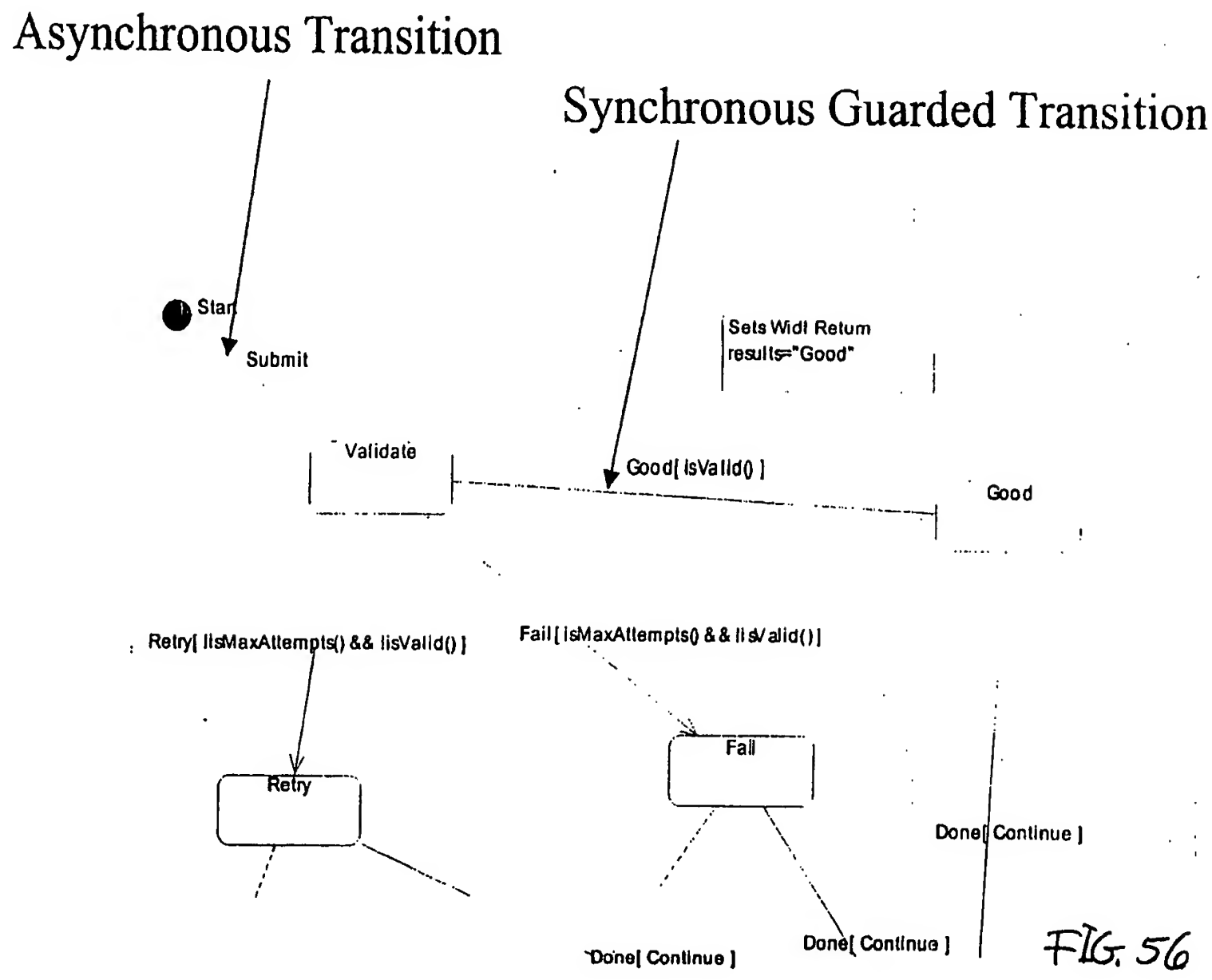


FIG. 56

WO 01/08007
PCT/US00/20069
59/76

Setting Animation Points

– Within the state machine any transition can be identified within the UML diagram as an animation point. When the application is executed the UML document will be displayed selecting the animation location.

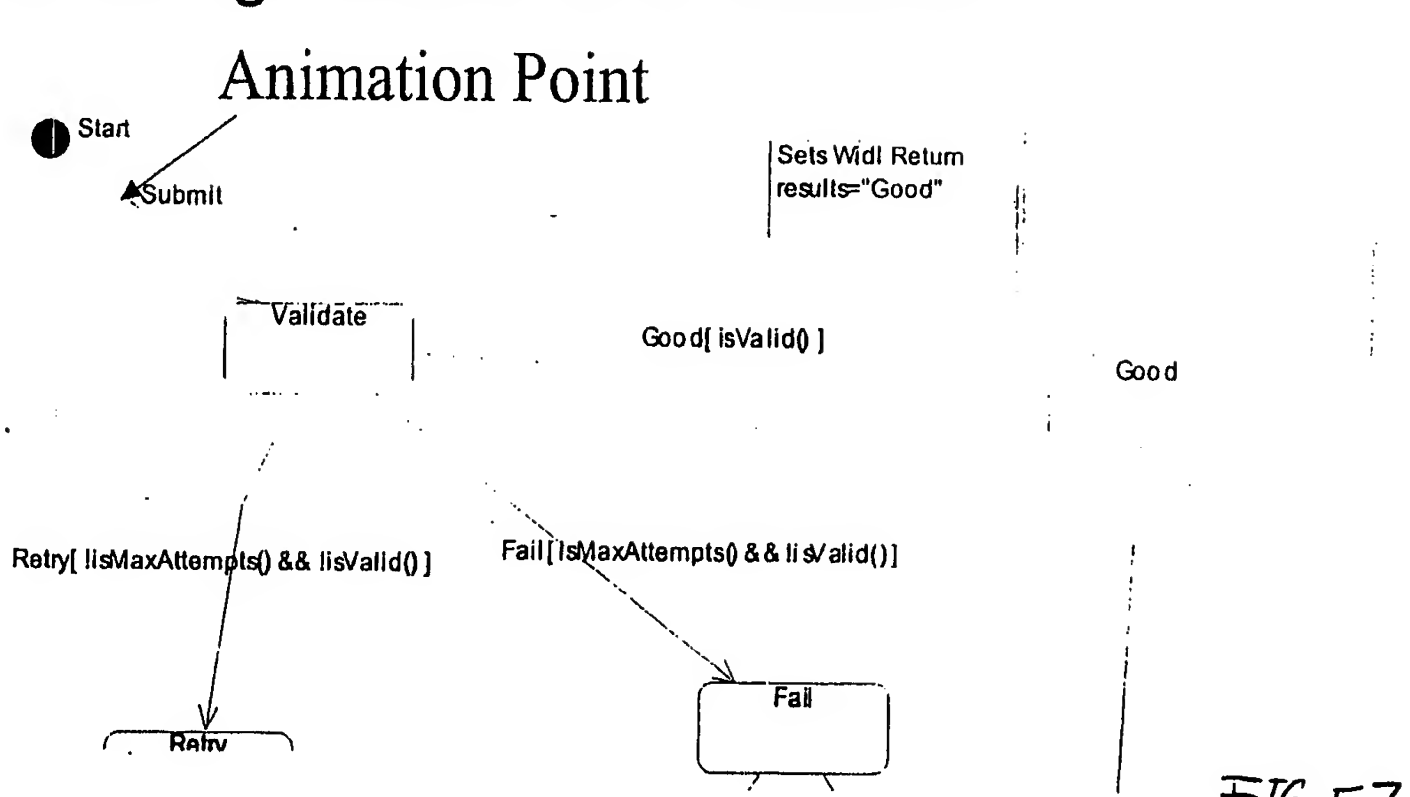


FIG-57

WO 01/08007
60/76
PCT/US00/20069

Activity Diagrams

– Activity Diagrams implement the behavior of algorithms for UML class specification methods.

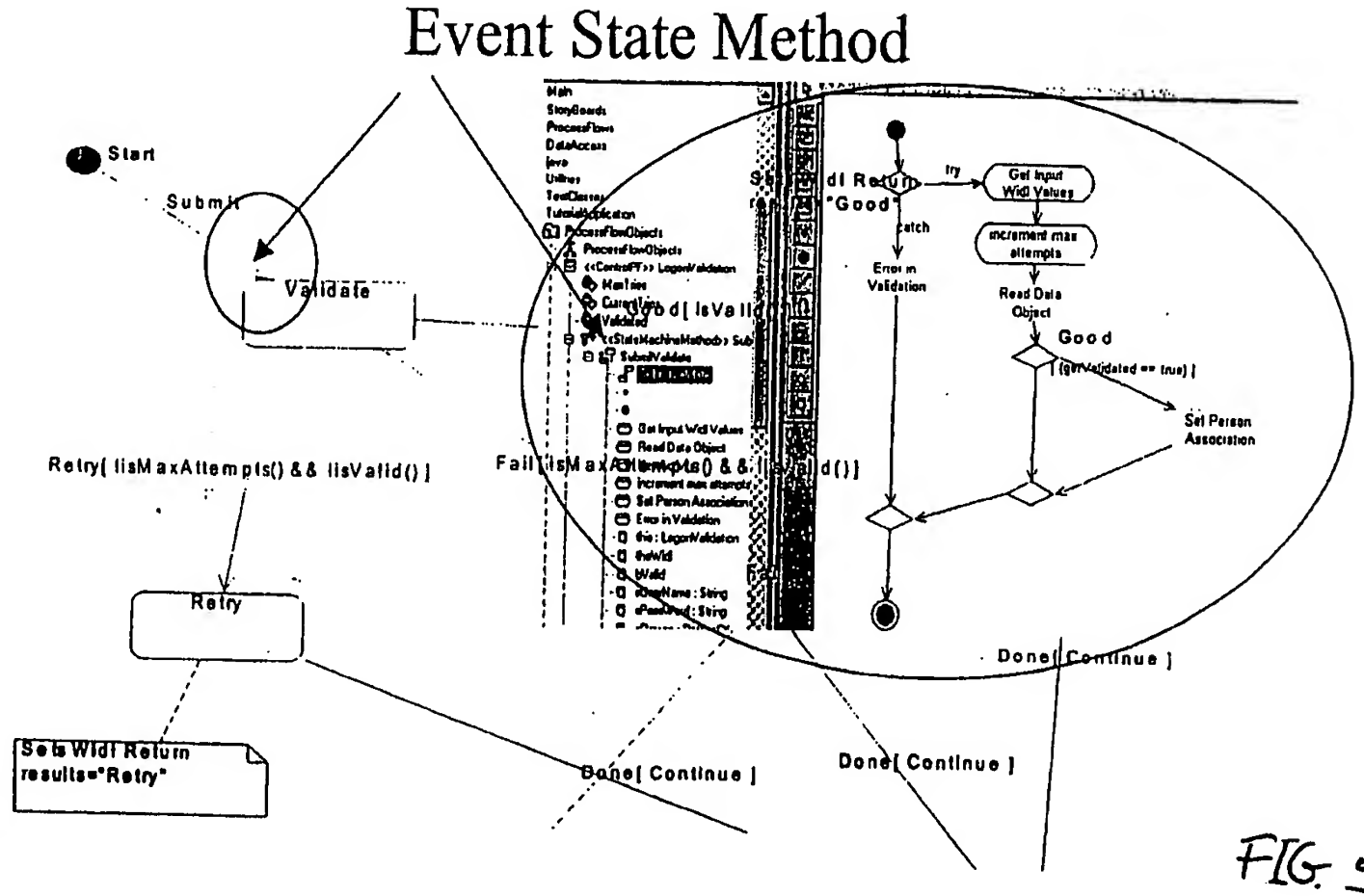


FIG. 58

UML Object Navigation Notation

– Activity Diagrams are represented in UML Object Navigation Notation. This notation manipulates the objects with their public attributes and methods. Using the object notation and Activity Editor within UML Factory you can define the complete behavior for a method implementation.

```
sUserName = theWidl.getInputParameter( "UserName" )
sPassWord = theWidl.getInputParameter( "Password" )
this.setValidated( false )
```

FIG. 59

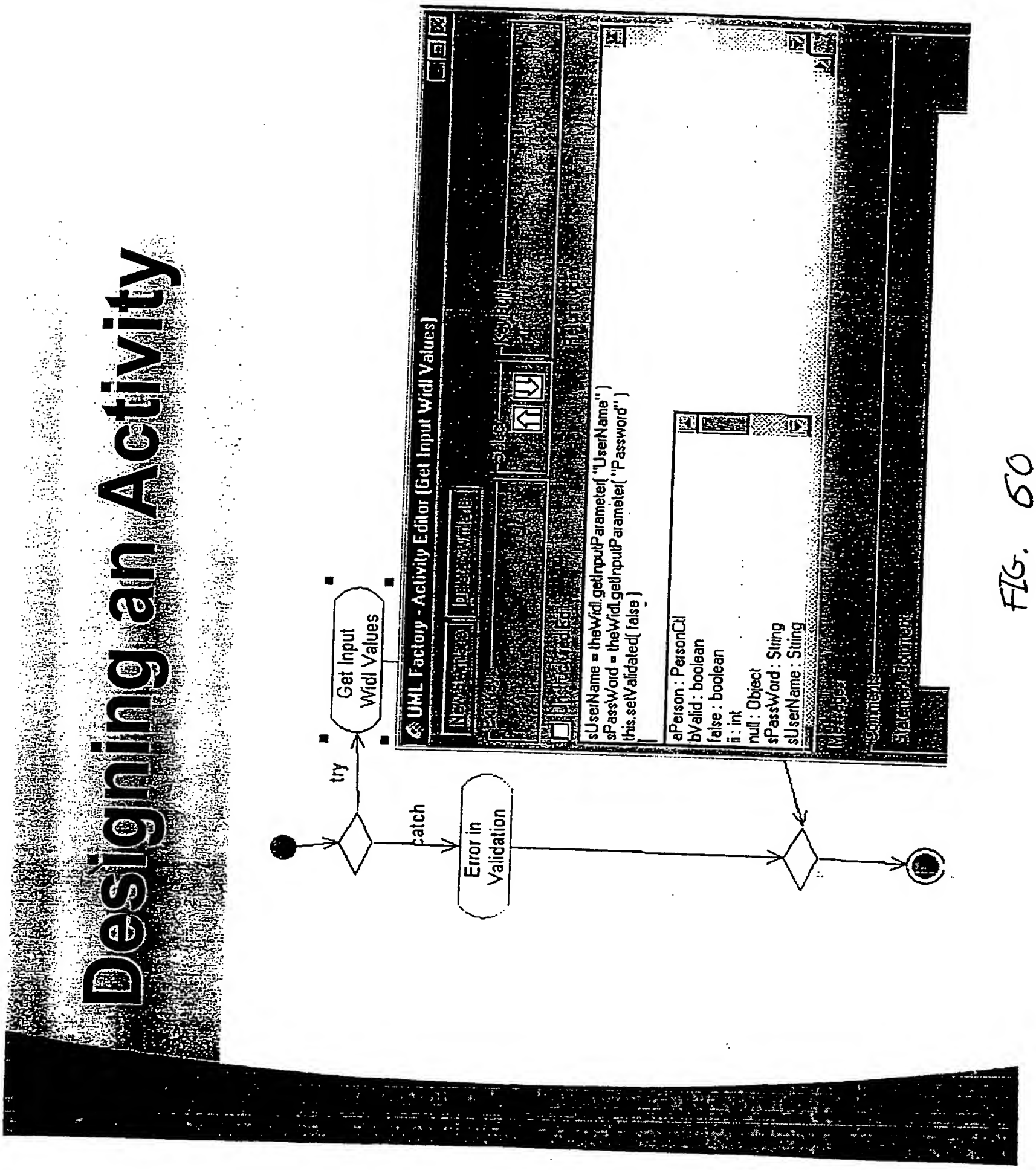

Designing an Activity
try
Get Input Widl Values
catch
Error in Validation
UML Factory - Activity Editor [Get Input Widl Values]
sUserName = theWidl.getInputParameter("UserName")
sPassWord = theWidl.getInputParameter("Password")
this.setValidated(false)
aPerson : PersonCtl
bValid : boolean
false : boolean
ii : int
null : Object
sPassWord : String
sUserName : String
Comments
Statement Element

FIG. 60

Accessing a Data Base

– Through the activity diagram implementations we can manipulate our database objects, defining the operations for all our database elements as required.

```
aPerson.PersonCtl(  )
aPerson.setFirstName( sUserName )
aPerson.setPassword( sPassWord )
this.setValidated( aPerson.load(  ) )
```

FIG. 61

WO 01/08007
PCT/US00/20069

Manipulating XML

– Within the activity diagrams we also define the XML manipulation for retrieving and building the XML documents going between the process flow logic and user interface elements.

```
sValue= new String( "PassageSoftware" );
aXMLMap.insert( this.getXML_CompanyName( ), sValue );
sValue= new String( "Richmond, VA" );
aXMLMap.insert( this.getXML_CompanyCity( ), sValue );
aXMLMap.insert( this.getXML_Person_NODE( ), null );
sName= new String( "Dick" );
sLName= new String( "Douglas" );
aXMLMap.insert( this.getXML_FirstName( ), sName );
aXMLMap.insert( this.getXML_LastName( ), sLName );
```

FIG. 62

WO 01/08007
PCT/US00/20069
65/76

Intelligent Advisor Code view

– The UML Factory Intelligent Adviser allows us to view the code generation results while designing the UML activity diagrams.

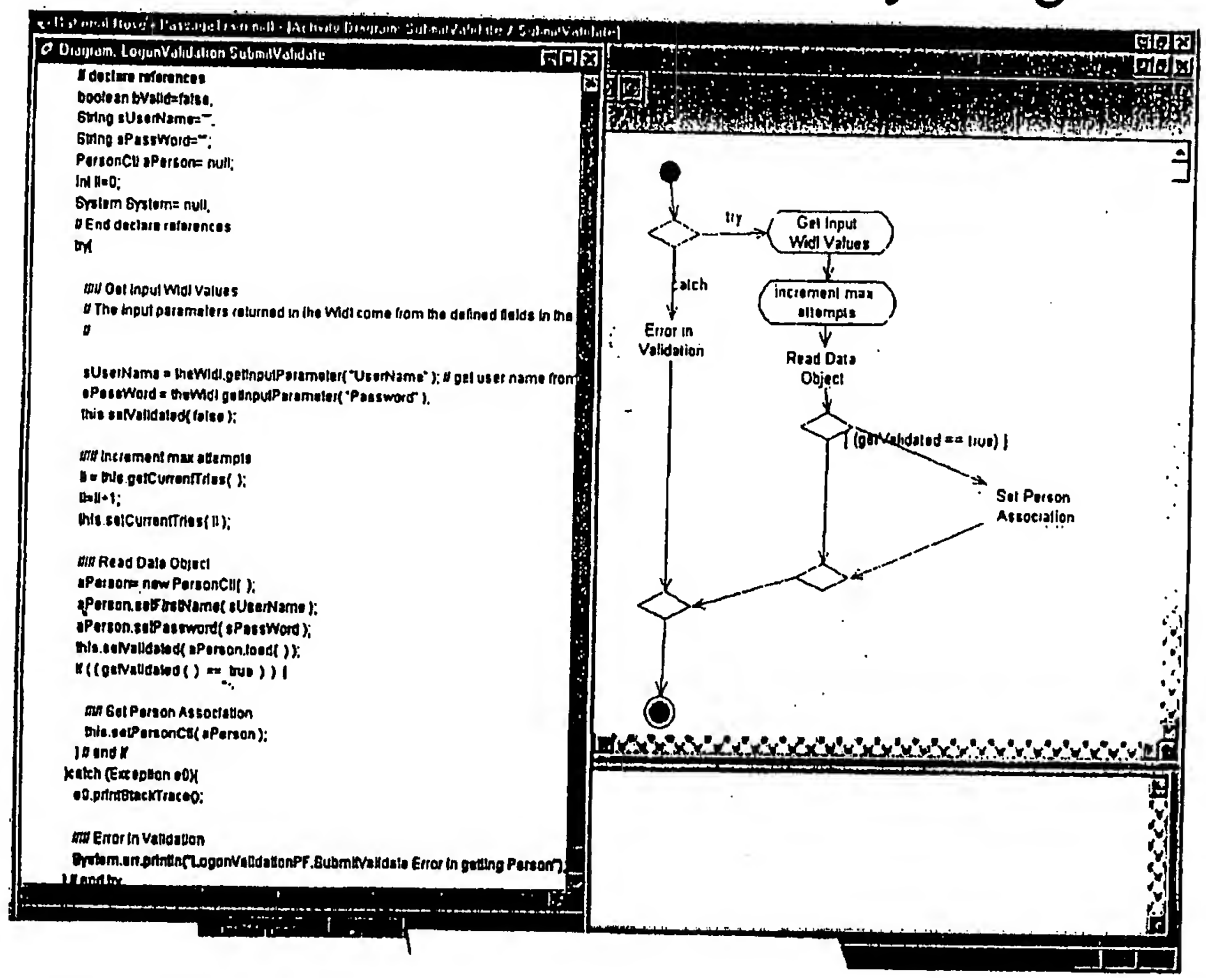
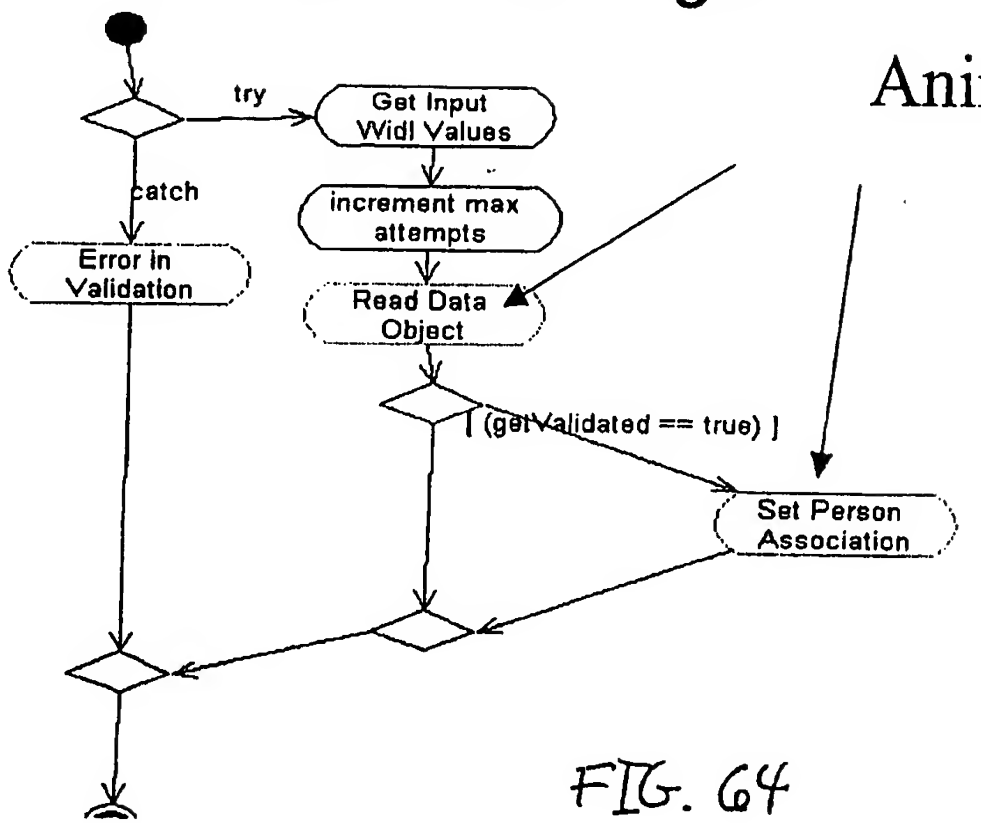


FIG. 63

WO 01/08007
PCT/US00/20069

Setting Animation points

– Any activity within the activity diagrams can also have animation points defined. When the application is executing the activity diagram will display and select the activities with animation settings.



Animation Points

FIG. 64

Generating the Application

– After the static and dynamic elements of our application are defined we can organize these elements into UML Component Diagrams to define deployment definitions. From the components definitions we can then generate the complete Java class code, compile the code and package it into the appropriate application JAR.

FIG. 65

WO 01/08007
PCT/US00/20069

Collaboration Diagram XML Generation

– The collaboration diagram information for our use case definitions is also generated into an XML description document identifying the interface elements, XML mappings, and control process flow information. This XML document then becomes the basis for execution through the application.

- The XML document defines the process flow through the application.
- The engine that iterates the collaboration diagram XML document was also designed and generated from a UML model.

FIG. 66

WO 01/08007
PCT/US00/20069
69/76

Control Process Flow Generation

– Dynamic behavior represented within an application is designed within a control process flow class specification. The following sections will briefly illustrate the generation processing capabilities for dynamic behavior within a UML environment. The following code examples were completely generated from the UML model.

FIG. 67

WO 01/08007
70/76
PCT/US00/20069

Generating the UML Class Specification

- *Looking at the State Machine implementation*
  - Current State
  - Incoming Event
  - Synchronous Transitions
  - Asynchronous Transitions
  - Start and End States
- *Packages, imports, declarations*
  - Java package specifications can be explicitly or implicitly defined by the class locations within the UML model.
- *Association and Attribute implementations*
  - Association implementations generate both the container and access methods for the stereotyped association. The type of container and resulting access methods are defined by the stereotype applied to the association between class specifications.

FIG. 68

WO 01/08007
PCT/US00/20069

Method Implementation

- *Activity Diagram Method Implementations*
  - Activity diagrams are generated following the UML Object Notation. The activity generation can be viewed with either as a fully generated class or using the Intelligent Advisor code window.
- *Round Trip engineering*
  - Roundtrip engineering allows developers to hand edit code generated from the UML Diagram. By Identifying elements which have been hand edited, the UML Factory generator will leave the method implementation as edited without forward generating from the model. The developer controls the granularly of roundtrip engineering.
- *UML Model Animation code*
  - Animation points placed within the code can be turned off for individual points, the entire model being generated and also ignored, from run time configuration properties

FIG-69

WO 01/08007
PCT/US00/20069
72/76

Executing the Architectures

– The following demonstrations will walk through executing the various architectures from the complete components on different types of application servers.

FIG. 70

Sample JSP™ Client to J2EE™ Application Container

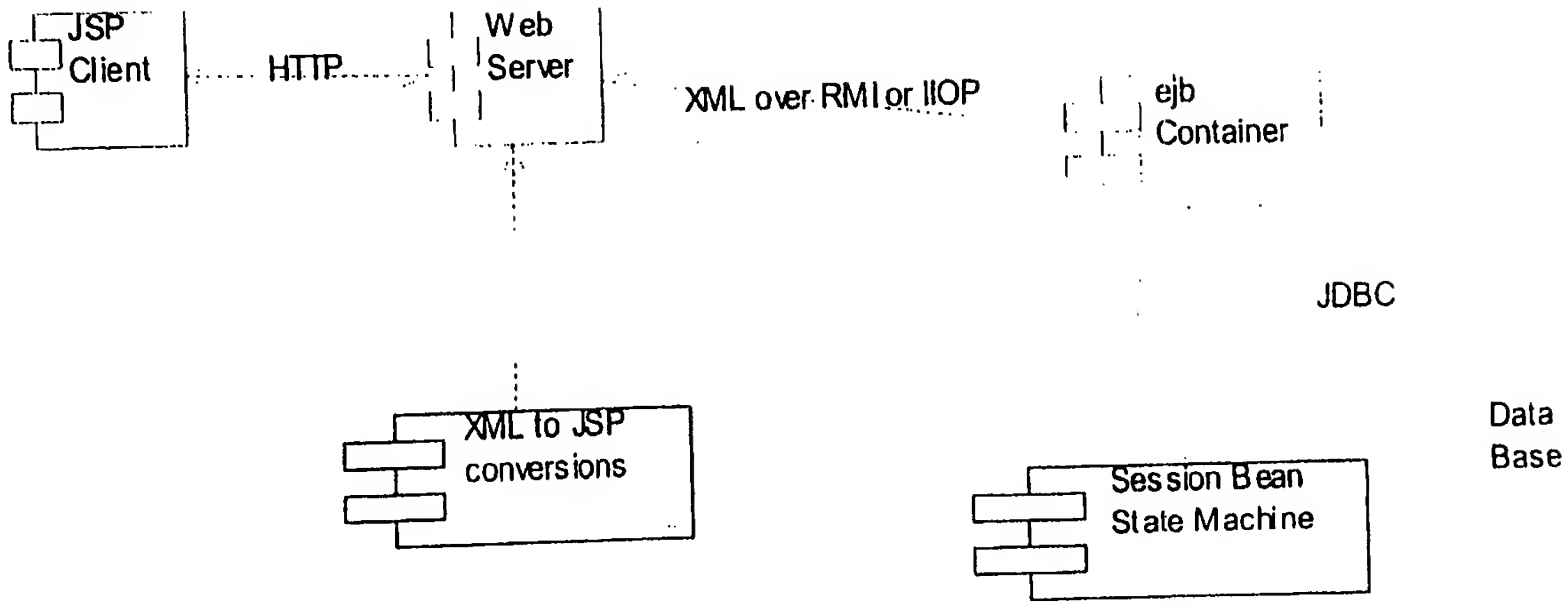


FIG. 7I

Summary

- This concludes the presentations and demonstrations for building n-tier enterprise applications with UML and XML data representations.
- The demonstrations have illustrated the ability to completely model both static and dynamic application behavior within a UML model.
- The UML model also defined components and deployment into multiple configurations.
- The modeled demonstration illustrated the ability to use the same components within different J2EE™ platform-enabled implementations on multiple architectural environments.

FIG. 72

WO 01/08007
PCT/US00/20069
75/76

Enterprise Development Benefits

- Enterprise computing can improve the software development process by combining the descriptive power of the UML notation, and the flexible data representation and distribution capabilities of XML with the object oriented, network, and portable capabilities of Java technology.
- Combining these technologies enables organizations to manage components increasing reuse, visibility, and implementation understanding; effectively reducing the complexity and total cost of ownership for deploying n-tier enterprise applications

FIG. 73

INTERNATIONAL SEARCH REPORT

Internal Application No
PCT/US 00/20069

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/44

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, INSPEC, IBM-TDB

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5 455 952 A (GJOVAAG INGHARD J) 3 October 1995 (1995-10-03) column 2, line 61 -column 3, line 13 column 10, line 23 - line 36 --- -/--	1-13

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
28 November 2000	05/12/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL – 2280 HV Rijswijk Tel. (+31-70) 340-2040. Tx. 31 651 epo nl. Fax: (+31-70) 340-3016	Brandt, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Interna[illegible]al Application No
PCT/US 00/20069

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	TOSHIMI MINOURA ET AL: "STRUCTURAL ACTIVE OBJECT SYSTEMS FOR SIMULATION" ACM SIGPLAN NOTICES,US,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 28, no. 10, 1 October 1993 (1993-10-01), pages 338-355, XP000411736 ISSN: 0362-1340 page 340, left-hand column, line 32 -right-hand column, line 2 page 341, left-hand column, line 24 -right-hand column, line 35 page 352, right-hand column, line 5 - line 15 ---	1-13
A	WO 97 35254 A (MASSACHUSETTS INST TECHNOLOGY) 25 September 1997 (1997-09-25) page 2, line 6 -page 6, line 17 -----	1-13

Form PCT/ISA/210 (continuation of second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Information on patent family members

Internal Application No
PCT/US 00/20069

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5455952 A	03-10-1995	NONE	
WO 9735254 A	25-09-1997	CA 2249386 A	25-09-1997
		EP 0888585 A	07-01-1999

(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

CORRECTED VERSION

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
1 February 2001 (01.02.2001)

PCT

(10) International Publication Number
WO 01/008007 A1

(51) International Patent Classification[7]: G06F 9/44

(21) International Application Number: PCT/US00/20069

(22) International Filing Date: 24 July 2000 (24.07.2000)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:

60/145,051	22 July 1999 (22.07.1999)	US
60/212,841	21 June 2000 (21.06.2000)	US

(71) Applicant: **PASSAGE SOFTWARE, LLC** [US/US]; 4120 Cox Road, Glen Allen, VA 23060 (US).

(72) Inventor: **HOWERY, William, D.**; 2715 Spinnaker Court, Richmond, VA 23233 (US).

(74) Agents: **MCNAMARA, Brian, J.** et al.; Foley & Lardner, Suite 500, 3000 K Street, N.W., Washington, DC 20007-5109 (US).

(81) Designated States *(national)*: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

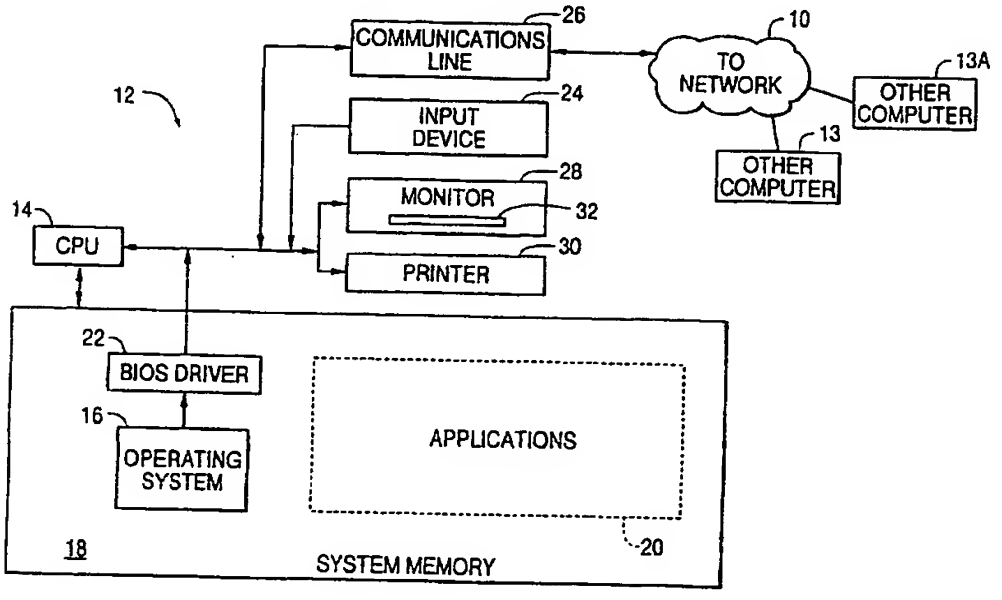
Published:
— *with international search report*

(48) Date of publication of this corrected version:
25 July 2002

(15) Information about Correction:
see PCT Gazette No. 30/2002 of 25 July 2002, Section II

[Continued on next page]

(54) Title: METHOD AND SYSTEM OF AUTOMATED GENERATION OF PROGRAM CODE FROM AN OBJECT ORIENTED MODEL



(57) Abstract: A computer implemented method of generating program code from user requirements based modelling diagrams of an object oriented (OO) model including the steps of defining a respective collaboration diagram for each interaction between two objects of the modelling diagram; defining interface, control and entity tier objects based on the modeling diagrams and the collaboration diagram; defining business rule control objects for encapsulating problem domain business rules; defining program flow control objects for encapsulating processes based on interactions; using standardized markup language for communicationg between the business rule control objects and the program flow control objects; and generating program code from the defined interface objects, control objects, entity objects, program flow control objects and business rule control objects.

WO 01/008007 A1

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

METHOD AND SYSTEM OF AUTOMATED GENERATION OF PROGRAM CODE FROM AN OBJECT ORIENTED MODEL

This application claims the benefit of priority under 35 U.S.C. 119(e) of provisional applications 60/145,051 filed on July 22, 1999, and 60/212,841 filed on June 21, 2000. The contents of both of these provisional applications (including their appendices) are incorporated herein in their entireties.

BACKGROUND OF THE INVENTION

Field of the Invention

This invention relates generally to the field of automatically generated program code and more particularly to a method and system of automated generation of program code for an object oriented model represented in Unified Modeling Language ("UML") notation.

Background of the Related Art

Current object oriented modeling tools provide the components for modeling the object oriented systems. In addition, these tools often provide limited capabilities for generating some aspects of the program code for these object oriented systems that implement the model. However, existing implementation capabilities of these object oriented modeling tools do not provide the complete end-to-end automated program code generation that completely implements the object oriented model.

Furthermore, since existing tools are not capable of generating the end-to-end code to implement a modeled system they are not able to maintain the integrity between the model and the program code. Accordingly, changes at the program code level have to be mapped back to the model in a manual process which gives rise to errors and also causes inconsistency between the model and the implemented code.

For example, the Rational Rose modeling tool ("Rose") provided by Rational Software Corporation, provides basic SQL data access capabilities, including select, insert, update and delete which can be implemented as Visual Basic code. However, the more

complex query functions require the creation of views external to the Rose tool. Coordinating such external data models with the Rose model gives rise to significant challenges and problems.

Furthermore, as discussed above, although Rose provides code generation capabilities for both Visual Basic and C++, considerable amount of hand coding is required to complete the implementation. Rose creates references to associated objects, references and initializes class attributes, declares class methods and implements prototype (stub) code with return data and input parameter types specified. However, the actual program code to complete the methods must still be produced outside of the Rose tool.

Business rules are defined within the Rose system. However, even though the business rules are defined within Rose, no rigorous procedure is provided within Rose to implement these business rules. That is, Rose does not provide a rigorous method for defining business rules so that they can be automatically implemented as program code by the Rose system.

In this context, it should be noted that UML is a general purpose notational language for specifying and visualizing complex software, especially large, object oriented software projects. UML builds on previous notational methods such as Booch, OMT, and OOSE. The UML is rapidly becoming a standard for object-oriented notations under the auspices of the Object Management Group (OMG). UML seeks to provide a common metamodel and a common notation so as to facilitate the development of systems on an architectural scale. Details of the OMG, UML and a Unified Method of system modeling and development based on UML are disclosed on the Internet, for example, at the following URL's www.omg.org and www.omg.org/uml.

As a background to understanding the present invention, a fundamental aspect of object oriented programming is that objects can be organized into classes in a hierarchical fashion and that the objects are interpretable. Classes are abstract generic descriptions of objects and their behaviors. Therefore, a class defines a certain category of methods and data within an object that belongs to that class. Methods comprise the procedures or code that implement the behaviors of the class and operate on the data within the class. Refinement of the methods of a generic class can be implemented by the creation of sub-classes which inherit the behaviors of its parent classes, from which they depend. In addition, the sub classes can have can have behaviors which originate at the sub class or modify the behaviors of its parent classes.

An instance of a class or an object is a specified individual entity that has an observable behavior. That is, an instance is a specific object characterized by having the behaviors defined by its class. Therefore, instances or objects are created and destroyed dynamically while the class is the abstract category under which the instance or object belongs. The instance or object inherits all the methods of its class and its data types but has particular individual values associated with it that are unique. Physically, there is only one location in a computer memory for a class whereas there may be numerous objects or instances of that class each of which has different values and different physical locations in memory.

SUMMARY OF THE INVENTION

Therefore, it is a general object of the invention to alleviate the problems and shortcomings identified above.

One of the objects of the invention is to provide a computer implemented method of generating program code for an object oriented (OO) model in a target environment in which the syntax for the data type primitives is generated and these generated data types are used in generating program code for the OO model.

Another one of the objects of the invention is provide a computer implemented method of generating program code from the modeling diagrams of an OO model.

Another object of the invention is to provide a computer implemented method of depicting asynchronous and synchronous events in a single state diagram of an OO model.

A further object of the invention is to provide a computer implemented method for an OO model in which the program code for a method is synchronized to the activity diagram of that method.

Another object of the invention is to provide a computer implemented method of generating context sensitive help for any element in an activity diagram of an OO method of an object in which context sensitive help is provided based on the context of the object.

These and other objects are achieved by providing a computer implemented method for generating program code for an Object Oriented (OO) model in a target environment including the steps of: defining abstract data type primitives for elements in an activity diagram of an Object Oriented (OO) method; generating the syntax for the abstract data type primitives in the target environment; and generating program code in the target environment, using the generated syntax for the abstract data types, for the OO method

depicted in the activity diagram.

Also provided is a computer implemented method of generating program code from user requirement based modeling diagrams of an object oriented (OO) model including the steps of: defining a collaboration diagram depicting the interaction between two artifacts of the modeling diagrams; defining interface, control and entity tier objects based on the modeling diagrams and the collaboration diagram; defining business rule control objects for encapsulating problem domain business rules; defining program flow control objects for encapsulating processes based on interactions; using standardized markup language for communicating between the business rule control objects and the program flow control objects; and generating program code from the defined interface objects, control objects, entity objects, program flow control objects and business rule control objects.

Also provided is a computer implemented method of depicting asynchronous and synchronous events in a single state diagram of an OO model including the steps of defining a guard condition constraint on an event transition between two states; determining an event transition as being synchronous if a guard condition is present; and determining an event transition as being asynchronous if a guard condition is not present.

Further provided is a computer implemented method of synchronizing program code for an Object Oriented (OO) method to the activity diagram of the OO method in an OO model including the steps of correlating each elements of the activity diagram to at least one line of program code; scanning the program code to identify lines of program code not correlated to any element of the activity diagram; inserting an element in the activity diagram corresponding to respective identified lines of program code not correlated to any elements of the activity diagram.

Also provided is a computer implemented method of generating context sensitive help for any element in an activity diagram of an Object Oriented (OO) method attached to an object in an OO model, including the steps of: determining the element of the activity diagram pointed to by a pointing device; scanning a local namespace available within the object to which the OO method is attached; and generating the context sensitive help based on the element of the activity diagram pointed to and the scanned local namespace of the object.

BRIEF DESCRIPTION OF THE DRAWINGS

The accompanying drawings, which are incorporated in and constitute a part of the

specification, illustrate a presently preferred embodiment of the invention, and, together with the general description given above and the detailed description of the preferred embodiment given below, serve to explain the principles of the invention.

Fig. 1 is schematic diagram illustrating a computer system suitable for implementing the present invention.

Fig. 2 is a schematic diagram illustrating a computer network that could be used to connect computer systems such as that illustrated in Fig. 1.

Fig. 3 shows a use case diagram for a Logon process use case.

Fig. 4 illustrates a collaboration class object diagram.

Fig. 5 illustrates a collaboration diagram.

Fig. 6 illustrates a control entity class diagram showing peer relationships.

Fig. 7 illustrates a class diagram demonstrating control entity data access showing relationships between database control classes.

Fig 8 illustrates exemplary data access code generated by the present invention.

Fig. 9 shows sample generated Java code implementing a mapping from a logical entity to a physical database.

Figs. 10a and 10b show a sample Java control schema generated according to the present invention.

Fig. 11 illustrates a class diagram for control program flow and control business rule classes.

Fig. 12 illustrates a sample state diagram for VRULogionPF control program flow class object.

Figs 13a-13c show sample code for a generated state machine.

Fig. 14 illustrates an activity diagram for an event/state intersection point.

Fig. 15 shows exemplary code for a generated method corresponding to the scenario diagram of Fig. 14.

Fig. 16 illustrates an exemplary XML object diagram.

Fig. 17 illustrates a generated XML schema.

Fig. 18 illustrates a generated XML DTD.

Figure 19 illustrates a template for a data access.

Figure 20 illustrates a resulting class specification from the merging of a template and a pattern.

Figure 21 shows additional examples of merging templates and patterns to generate

class specifications.

Figures 22-73 disclose the details of one preferred embodiment of the present invention.

DESCRIPTION OF THE PREFERRED EMBODIMENT(S)

With reference to the figures, Figure 1 shows a block diagram showing the components of a general purpose computer system 12 connected to an electronic network 10, such as a computer network. The computer network can also be a public network, such as the Internet, a private network or a virtual private network. As shown in the figure 1, the computer system 12 includes a central processing unit (CPU) 14 connected to a system memory 18. The system memory 18 typically contains an operating system 16, a BIOS driver 22, and application programs 20. In addition, the computer system 12 contains input devices 24 such as a mouse and a keyboard 32, and output devices such as a printer 30 and a display monitor 28.

The computer system generally includes a communications interface 26, such as an ethernet card, to communicate to the electronic network 10. Other computer systems 13 and 13A also connect to the electronic network 10 which can be implemented as Wide Area Network (WAN) or as an inter-network such as the Internet. One of skill in the art would recognize that the above system describes the typical components of a computer system connected to an electronic network. It should be appreciated that many other similar configurations are within the abilities of one skilled in the art and all of these configurations could be used with the methods of the present invention. Furthermore, it should be recognized that the computer system and network disclosed herein can be programmed and configured, by one skilled in the art, to implement the method steps discussed further herein.

In addition to being operable in a single computer system, the present invention may also be implemented in a networked computer system. An example of a typical computer network is shown in Fig. 2 which depicts a plurality of workstations 140 connected directly or through a host server 142 to a client/server network 144. The client/server network may include an object oriented messaging system , such as the Object Management Group's Common Object Request Broker (CORBA) or Microsoft's Component Object Model (COM) for controlling the communication between distributed objects across clients and servers. The client/server computers could be connected using a

standard ethernet bus although it is to be understood that all other types of networks, including wireless networks, could also be used in implementations of the present invention.

One aspect of the present invention provides the generation of data access classes from entity and control classes. A prior art modeling tool, such as Rational Rose, provides basic SQL data access capabilities including select, insert, update and delete realized as Visual Basic code. More complex data query functions require the creation of views outside of Rose. Coordinating an external model with the Rose model creates significant challenges. The present invention provides a two-tier data model having an entity tier and a control tier. The entity tier captures the physical and logical relationships. The control tier tracks the relationships between objects. This allows the data modeling to be done independent of business process constraints. The data access functionality can be generated in several programming languages such as Java and Visual Basic. If Visual Basic is the target, joins can be modeled without the creation of separate views. The more sophisticated capabilities of Java allow for the database schema and the table relationships from the control tier to be incorporated directly into the implementation code. A business object model can then use this code to generate complex queries spanning multiple tables.

A preferred embodiment of the present invention will be described next. It should be understood that the following description describes the preferred embodiment of the invention and is not intended to be limitative of the invention. Therefore, the preferred embodiment uses UML terminology and modeling diagrams implemented by Rose to illustrate the preferred embodiment. However, the present invention is intended to apply to all object oriented modeling and development systems that use modeling diagrams and notation terminology that is equivalent to that discussed below with respect to the preferred embodiment.

Fig. 3 shows a use case diagram for a logon process use case. The use case diagram is an example of a use requirement based modeling diagram or construct. In the following description, the "logon" process is described because it is representative of the processes that may be implemented using the present invention. A member of the public (actor) 200 starts a logon process to, for example, access a system that requires a log-in. The logon interaction 205 defines an interaction between the actor and a Logon process 210 that implements the login to the system.

The present invention requires that each interaction must have a collaboration

diagram defining the action and the attributes involved in that action. A collaboration diagram specifies the tangible artifacts defining the interaction between a use case process and an actor or with other processes. Therefore, the preferred embodiment of the present invention requires a one-to-one relationship between the interaction and the collaboration diagrams. In this context, tangible artifacts are defined and measurable pieces of information being sent from a use case process to the actor or from the actor to the use case process. Therefore, artifacts represent the information through to the system.

Creation of a collaboration diagram requires the definition of objects and their attributes within a class diagram. Fig. 4 illustrates a collaboration class object diagram that defines the objects and their attributes that are necessary for the collaboration diagram that is illustrated in Fig. 5. Therefore, Fig. 4 contains exemplary UML class specifications with the Storyboard Stereotype. A Storyboard Stereotype identifies user interface elements. The attributes and attribute type in the class specification define the types of artifacts used with the user interface pages represented by a storyboard stereotype class specification. Therefore, the objects 401, 402, 403, and 404 shown in Fig. 4 correspond to elements 501, 502, 503, and 504, respectively, in the exemplary collaboration diagram shown in Fig. 5

The collaboration diagram illustrated in Fig. 5 depicts the logon interaction 205 identified in the use case diagram shown in Fig. 3. The collaboration diagram produced through this process is tightly coupled to both the use case (shown in Fig. 3) and the objects defined in the class diagram (shown in Fig. 4). Collaboration diagrams contain references place holders for user interface objects. The user interface objects (Storyboards) contain the artifacts or abstract data elements that a user will either send or receive from the system. These events are modeled into a process flow that identifies which additional user interface or logical domain objects will receive the users input. Logical domain objects also respond with result events that trigger transitions to other interface or domain objects within the collaboration diagram.

The next step in the present invention involves data modeling of the persistent and tangible artifacts which are artifacts that are stored in a database or other long term storage. In the prior art, Ivar Jacobson defined three primary system partitioning stereotypes: interface; control; and entity. The entity tier (or stereotype) related to the persistent data objects, the control tier to the mid-tier processes, and the interface tier to the interactions with the users or other systems. The present invention provides two

additional tiers: a control program flow tier and a control business rules tier. Both of these additional tiers provide additional definition to the mid-tier processing and are discussed in more detail further herein.

Therefore, the present invention provides that two class diagrams are developed: (i) an Entity Relationship Diagram (ERD) class diagram defining the entity and the control tiers (similar to Jacobson's Objectory process); and (ii) a control program flow/business rules class diagram. Fig. 6 shows an exemplary control entity class diagram showing peer relationship between two objects, 601 and 602.

The present invention provides that once the control stereotype objects and their relationships are defined, as discussed above, the peer relationships and the entity models for the Entity Relationship Diagrams are automatically generated. Therefore, the present invention provides that the program code implementing the invention understands the relationship between the control data access and the entity persistent data access classes. When this relationship is created the program code automatically sets the relationships required for generating the complete program implementation for the control and entity data access. This automatic generation provides consistency, completeness, and reduces a significant amount of tedious and redundant effort in prior art systems.

The present invention also generates a class diagram demonstrating control entity data access showing the relationships between Database control classes as shown, for example, in Fig. 7. Fig. 7 illustrates an exemplary set of data access classes 701-705 with their interconnecting relationships. The present invention understands the attributes of the data access classes through the schema information extracted from the UML model in the entity and control class specifications. The invention provides for propagating the role association names identifying the data elements of the control data access specifications that relate each class to other classes. The present invention also creates program code operation methods for implementing run-time navigation between control data access class specifications.

The present invention then provides for creating the code for data access. A generator processes each entity class using the properties as defined by the processes discussed above. It creates a metastructure defining a mapping between class and class attributes to database name and columns. To define this mapping , the present invention provides for using the semantic naming applied to the UML model. With this semantic meaning, the present invention understands how each attribute of the entity class

specification will be implemented into a persistent data base environment. This semantic information is interpreted in program code information which automates how all objects are manipulated in run-time programs. This mapping is generally Interface Definition Language (IDL) specific. This process implements an interface (presently in Java and COM) that the business database object model can use to interpret the class diagrams for managing data access. Exemplary Java program code that implements select/update/delete and select loading of a collection is shown in Fig. 8.

The present invention then generates the control stereotype tier. This is done by cycling through the class diagram, picking out each class stereotype control and associated peer-relationship entity class. The control class uses the related entity class to perform data access management. Additionally, the database schema diagram is abstracted from the class diagram. This schema is then used to dynamically create complex query joins between multiple objects. Fig. 9 shows sample generated Java code implementing a mapping from a logical entity to a physical database. Figs. 10a-10b show a sample Java control schema generated in accordance with the present invention. The present invention uses the UML model that contains the relationships between persistent class specifications and also contains the information required for these specific class objects to access and manipulate information within a database. When the present invention creates the program code to access and manipulate information within a database, it interprets the UML model relationship information creating a schema or schematic representation of the UML model. This schema contains the knowledge required for the present invention to automatically transform relationship database information into object oriented run-time program structures.

The next step in the present invention requires the definition of the business rules. In the collaboration diagram illustrated in Fig. 5, a submit action 505 is shown that activates a business rule. A business rule is a process that evaluates business domain data and makes a decision accordingly. The present invention provides that business rules are defined under two separate stereotypes, either as control program flow or a control business rule stereotypes. The control business rules encapsulate the problem domain business rules. The control program flow objects control the path a transaction takes through the application based on the constraints provided by the business rules. These control program flow and control business rules can be defined in a class diagram. Each class contains a state diagram which defines events, states, and methods that belong to that

class.

Fig. 11 shows a class diagram for exemplary control program flow and control business rule classes 1101 and 1102, respectively. In addition, it shows the relationships between the control business rules and the database control classes. Fig. 12 shows a representative state machine. Therefore, each control process flow for control business rule object may have an associated state chart diagram. The present invention provides for translating this state chart diagram to a program code representation to manage events coming into the object or transitioning through the various states of the object. Figure 11 also illustrates a representative set of relationships between associated objects completing a system implementation. Typically, a control process flow object will relate to some domain business rule object. That control business rule object will then subsequently relate to database elements through the control data access stereotyped class specification.

Once the class diagrams and the state machines have been defined as discussed above, the present invention provides that the class code embodying the state machine that executes the business rule is generated. A state diagram may be associated with any control process flow or control business rule class specification. The state chart contains events transitioning between the defined states of that object. The present invention translates between these events and the guard conditions navigating boolean transitions between states into a code implementation that can receive synchronous and asynchronous events to implement the state transitions for a given object. Sample code for a generated state machine (shown in Fig. 12) for the VRULogonPF control program flow object is shown in Figs. 13a-13c.

It should be noted that the implementation code for class, business rules, and program flow can be used to generate standard container patterns (set dictionary lookup, hashtable, etc.) for managing associated objects. The abstraction of relationships to other objects facilitates the access and maintainability of complex object models.

In order to generate the class code for a state machine to function in a completed application, the present invention requires that the specific operations that take place at each intersection point between an event and a state be defined. This is accomplished though the use of activity diagrams, one of which must be associated with each event/state intersection point. Fig. 14 provides an example of an activity diagram for one of the intersection points between an event and a state shown in the state diagram of Fig. 12. This activity diagram corresponds to the intersection between the event = "TooMany" and

state = "Fail."

Fig. 15 shows exemplary generated code for the method of the VRULogonPF activity diagram (as shown in Fig. 14) for the event = "TooMany" and state = "Fail."

Therefore, the present invention extends beyond the conventional use of an activity diagram in a conventional tool, such as that provided by Rose, implementing UML. In the prior art, activity diagrams are used to graphically represent decisions and flows through an application. However, the prior art did not provide for the ability to define a behavioral scope of the activity diagrams, or simplify complex algorithms with many steps into their block representations of the activity diagrams. Furthermore, the prior art was unable to minimize detail required for representing swim lane objects in the activity. The present invention provides for assigning behavioral scope to the activity diagrams and provides for collapsing multiple complex steps into single activities. The present invention provides program code to automate work for a designer to manage multiple swim lanes and interactions between multiple swim lanes.

Therefore, one aspect of the present invention provides a specific one-to-one correlation between the event states, the class operations and activity diagrams. Therefore, by combining the business rules, modeled as described above, and the state machine activity diagrams, the present invention permits the generation of complete program code from an object model. Furthermore, another benefit of the present invention is that the information required for code generation is modeled in a specific sequence of modeling constructs and the program code can be generated from these modeling constructs in any target platform/language combination that supports the modeling constructs described above. Therefore, the modeling constructs are not specific to any single platform or language.

Another aspect of the present invention provides that cross tier communications can be accomplished by using the constructs of standardized markup language, such as XML (eXtensible Markup Language). Therefore, the present invention provides that the tiers of the completed application communicate through XML. This creates the challenge of managing free-form text information in a structured object oriented paradigm. The present invention addresses this challenge by extending UML onto the XML world. This is done by providing XML map extensions to the class diagrams. Therefore, the present invention provides that the XML stereotype classes and associations are set and the relationships between them are defined. Fig. 16 provides an examples of an XML object diagram

according to the present invention.

These XML maps are then related to a control program flow object as shown in Fig. 16. Therefore, when the present invention generates the application, it interprets the associated XML diagram to generate the Data Type Definition (DTD) and the XML schema used by the business object model. Fig. 17 provides an exemplary XML schema generated to correspond to the XML object diagram shown in Fig. 16. Fig. 18 shows the exemplary DTD generated from the XML map object diagram shown in Fig. 16.

Therefore, one aspect of the present invention provides for understanding and using the relationships between multiple objects in UML class diagrams. The invention then transforms these relationships into structural affirmations with schema information similar to database access representation. This structural schema information provides the information for program code (according to the present invention) to insert, extract, and navigate through XML documents. The structure of these XML documents can be defined through the UML model. The invention uses the XMLMap stereotype to represent a hierarchical structure for transformation into an XML map schema. The control process flow attached to the XML map structure receives the program code to define and implement the XMLMap manipulation and navigation program code.

One aspect of the present invention provides a method of synchronization of the program code that implements a method and an activity diagram that corresponds to that method. This is accomplished by correlating each element of the activity diagram to one or more lines of program code that implement the method. Furthermore, the sequence of elements in the activity diagram are also correlated to a sequence of the identified lines of program code that correspond to a particular elements.

Accordingly, if the program code is changed the method of the present invention provides that the program code is scanned to determine if any of lines of program code are not correlated to an element of the activity diagram. If any lines of uncorrelated program code are found, an activity diagram element that corresponds to the uncorrelated program code is inserted in the activity diagram. Since, the sequence of the elements of the activity diagram are also correlated to the sequence of the lines of program code, the position of the uncorrelated lines of code is used to determine the insertion point of the element in the activity diagram. In this way, the present invention synchronizes the program code that implements a method with the activity diagram corresponding to that method.

Another aspect of the present invention provides a computer implemented method of

generating context sensitive help for any element in an activity diagram of an Object Oriented (OO) method attached to an object in an OO model. The method involves determining the element of the activity diagram pointed to by a pointing device such as a cursor controlled by a mouse. Thereafter, the method of the present invention scans a local namespace available within the object to which the OO method is attached, and generates the context sensitive help based on the element of the activity diagram pointed to and the scanned local namespace of the object. This context sensitive help also builds the UML constructs that permit the generation of the program code that implements a method corresponding to an activity diagram.

A further aspect of the present invention provides a computer implemented method for generating the target code for data types in various different target environments. To this end, the present invention provides for the definition of abstract data type primitives for the data types. For example, these abstract data type primitives can be implemented as objects with data and methods (or behavior) corresponding to a particular data type. The present invention also provides for the generation of target code for these abstract data type primitives for a particular target environment. In this way, the present invention provides for modeling business objects and data types independent of the target environment.

Another aspect of the present invention provides a computer implemented method for depicting both synchronous and asynchronous events in a singe state diagram of an OO model. This is accomplished by interpreting an event having a guard condition constraint as defining a synchronous event transition whereas events that do not have guard condition constraints are defined as asynchronous event transitions. For example, with reference to Fig. 12, the guarded events 1201 and 1202 define synchronous event transitions while the event transitions 1203 and 1204 that do not have a guard condition define asynchronous event transitions. In this way, the present invention provides for the representation and interpretation of both synchronous and asynchronous events in one state diagram.

Another aspect of the present invention provides a computer implemented method and software for using template and pattern classes to generate a full class specification. Accordingly, the present invention provides that a class specification and activity details are generated from a tokenized UML template. Many software modules have precise patterns for implementation. These patterns have variations that are specific to particular instances of a given application.

In this context, it should be noted that in object-oriented programming, a pattern

can contain the description of certain objects and object classes to be used, along with their attributes and dependencies, and the general approach for solving a particular problem. A collection of patterns, called a pattern framework, can also be used to solve a specific problem. A book, "Design Patterns: Elements of Reusable Object-Oriented Software," by E. Gamma, R. Helm. R. Johnson, and J. Vlissides is crediting with creating an interest in design patterns in object-oriented programming.

The problem of database access, for example, implies a very rigid pattern on manipulating the database tables and fields. The variant components for a given class on a database access pattern, for example, would be the number of data columns and class attributes, as well as the names and types of these attributes. The data access pattern could utilize the same processes and procedures for reading, writing, creating and updating the class table object. The present invention provides for modeling the precise and constant elements of a design pattern, and for substituting at design time the variant elements of that particular design pattern. By utilizing template definitions within the class specifications allows the present invention to iterate a template specification over a specific instance that defines the variant elements of the pattern.

The present invention tokenizes the various static elements of a class specification and creates a new class specification merged from a template and a pattern. The template contains tokenized names identifying elements such as Class Name, Attribute Name, Association, operation and other static class items. The present invention provides a <<Pattern>> Stereotype Class Specification that contains the definitions for these variant token elements from the present invention's <<Template>> Stereotype Specification.

By utilizing a template Stereotype Class Specification and multiple Patterns for the variant elements, the present invention creates a complete Class Specification, with static and dynamic elements implemented per the Pattern associations within the variant template stereotype Class Specification.

Figure 19 is an example that demonstrates a Template 1901 for an EJB (Enterprise Java Bean) data access. The template 1901 is realized by a Pattern 1902, specifying the Class Name, Attributes, and Method names. The resulting class 2001 (shown in Fig. 20) is generated by the merging of the Template 1901 and the Pattern 1902.

As shown in figure 20, each tokenized element within the pattern specification is replicated for the template specification for each item defined in the pattern tokens. For

example: the attribute token is defined for the pattern set and get methods. According to the present invention, this instructs the present invention to create a get and set method for each attribute in the target template. The target template 2001 will define variant attributes and Data types for those attributes. The pattern specification will define a prototypical get and set method for attributes and the complete activity implementations for the attribute methods. When the present invention generates the complete class specification for this pattern and template relationship, it will substitute each attribute name and data type into the respective tokens for the prototypical methods defined within the pattern. The result is a complete class with the get and set methods for each attribute in the template. That is, the prototypical behavior is defined within the template and transferred to the pattern for each tokenized element. The pattern contains the data elements and/or methods with tokenized attributes or class definitions. When the template is applied to the pattern the dynamic behavior of the template will be transformed for each token element within the pattern.

Figure 21 is another example in which templates 2101 and 2102 are merged with one or more of the patterns 2103-2105 to generate class specifications 2106-2109. That is, class specification 2106 is generated by merging template 2102 with pattern 2103, class specification 2107 is generated by merging template 2102 with pattern 2104, class specification 2108 is generated by merging template 2101 with pattern 2105, and class specification 2109 is generated by merging template 2102 with pattern 2105.

According to the present invention, this pattern process can be implemented for any predictive and repetitive software pattern. A designer creates the predictive pattern with tokens representing the variant elements and then creates a template defining the variant elements. Then, the present invention provides for merging the pattern and template to create the complete class specification.

Figs. 22-73 disclose the details of one preferred embodiment of the present invention.

Other embodiments of the invention will be apparent to those skilled in the art from a consideration of the specification and the practice of the invention disclosed herein. It is intended that the specification be considered as exemplary only, with the true scope and spirit of the invention being indicated by the following claims.

What is claimed is:

1. A computer implemented method of generating program code for an Object Oriented (OO) model in a target environment comprising the steps of:

defining abstract data type primitives for elements in an activity diagram of an Object Oriented (OO) method;

generating the syntax for the abstract data type primitives in the target environment; and

generating program code in the target environment, using the generated syntax for the abstract data types, for the OO method depicted in the activity diagram.

2. A computer implemented method of generating program code from user requirements based modeling diagrams of an object oriented (OO) model comprising the steps of:

defining a respective collaboration diagram for each interaction between two objects of the modeling diagram;

defining interface, control and entity tier objects based on the modeling diagrams and the collaboration diagram;

defining business rule control objects for encapsulating problem domain business rules;

defining program flow control objects for encapsulating processes based on interactions;

using standardized markup language for communicating between the business rule control objects and the program flow control objects; and

generating program code from the defined interface objects, control objects, entity objects, program flow control objects and business rule control objects.

3. A computer implemented method of depicting asynchronous and synchronous events in a single state diagram of an OO model comprising the steps of:

defining a guard condition constraint on an event transition between two states;

determining an event transition as being synchronous if a guard condition is present; and

determining an event transition as being asynchronous if a guard condition is not

present.

4. A computer implemented method of synchronizing program code for an Object Oriented (OO) method to the activity diagram of the OO method in an OO model comprising the steps of:

correlating each elements of the activity diagram to at least one line of program code;

scanning the program code to identify lines of program code not correlated to any element of the activity diagram; and

inserting an element in the activity diagram corresponding to respective identified lines of program code not correlated to any elements of the activity diagram.

5. A computer implemented method of generating context sensitive help for any element in an activity diagram of an Object Oriented (OO) method attached to an object in an OO model, comprising the steps of:

determining the element of the activity diagram pointed to by a pointing device;

scanning a local namespace available within the object to which the OO method is attached; and

generating the context sensitive help based on the element of the activity diagram pointed to and the scanned local namespace of the object.

6. A computer readable data storage medium having program code recorded thereon for generating target program code for an object oriented model in a target environment, the program code comprising:

a first program code that allows the definition of abstract data type primitives for elements in an activity diagram of an object oriented (OO) method;

a second program code that generates the syntax for the abstract data type primitives in the target environment; and

a third program code that generates the target program code in the target environment, using the generated syntax for the abstract data types, for the OO method depicted in the activity diagram.

7 A computer readable data storage medium having program code recorded thereon for generating target program code from user requirements based modeling diagrams of an object oriented model, the program code comprising:

a first program code that allows the definition of a respective collaboration diagram for each interaction between two objects of the modeling diagram;

a second program code that allows the definition of interface, control, and entity tier objects based on the modeling diagram and the collaboration diagram;

a third program code that allows the definition of business rule control objects that encapsulate the problem domain business rules;

a fourth program code that allows the definition of program flow control objects for encapsulating process based on interaction;

a fifth program code that uses standardized markup language for communicating between the business rule control objects and the program flow control objects; and

a sixth program code that generates the target program code from the defined interface objects, control objects, entity objects, program flow control objects, and business rule control objects.

8. A computer readable data storage having program code recorded thereon for depicting asynchronous and synchronous events in a single state diagram of an OO model method, the program code comprising:

a first program code that allows defining a guard condition constraint on an event transition between two states; and

a second program code that determines the event transition to be synchronous of the guard condition is present and determines the event transition to be asynchronous if the guard condition is not present.

9. A computer readable data storage having program code recorded thereon for synchronizing generated program code for an object oriented (OO) method to an activity diagram of the OO method in an OO model, the program code comprising:

a first program code that correlates each element of the activity diagram to at least one line of program code;

a second program code that scans the generated program code to identify lines of generated program code that are not correlated to any element of the activity diagram; and

a third program code that inserts an element in the activity diagram that corresponds to respective identified lines of program code not correlated to any elements of the activity diagram.

10. A computer readable data storage having program code recorded thereon for generating context sensitive help for any element in an activity diagram of an object oriented (OO) method attached to an object in an OO model, the program code comprising:

a first program code that determines the element of the activity diagram pointed to by a pointing device;

a second program code that scans the local namespace available within the object to which the OO method is attached; and

a third program code that generates the context sensitive help based on the element of the activity diagram pointed to and the scanned local namespace of the object.

11. The method according to claim 2, wherein the step of using standardized markup language to communicate between objects uses the eXtensible markup language (XML) and includes the steps of:

developing an XML map extension to a class diagram to create an XML map object diagram that relates XML map objects to control program flow objects;

generating XML schema from the XML map object diagram; and

generating XML Data Type Definitions (DTD) from the XML map object diagram.

12. A computer implemented method of generating an object oriented class specification, the method comprising the steps of:

creating a template stereotype specification with tokens identifying variant class elements;

creating a pattern stereotype specification containing definitions for the variant class elements; and

merging the template stereotype specification and the pattern stereotype specification to generate a class specification wherein the tokens identifying the variant class elements are replaced by definitions for the variant class elements contained in the pattern stereotype specification.

13. The method according to claim 12, wherein the creating a pattern stereotype specification step includes creating a plurality of pattern stereotype specifications, and

wherein the merging step includes merging any one of the plurality of pattern stereotype specifications with the template stereotype specification to generate the class specification.

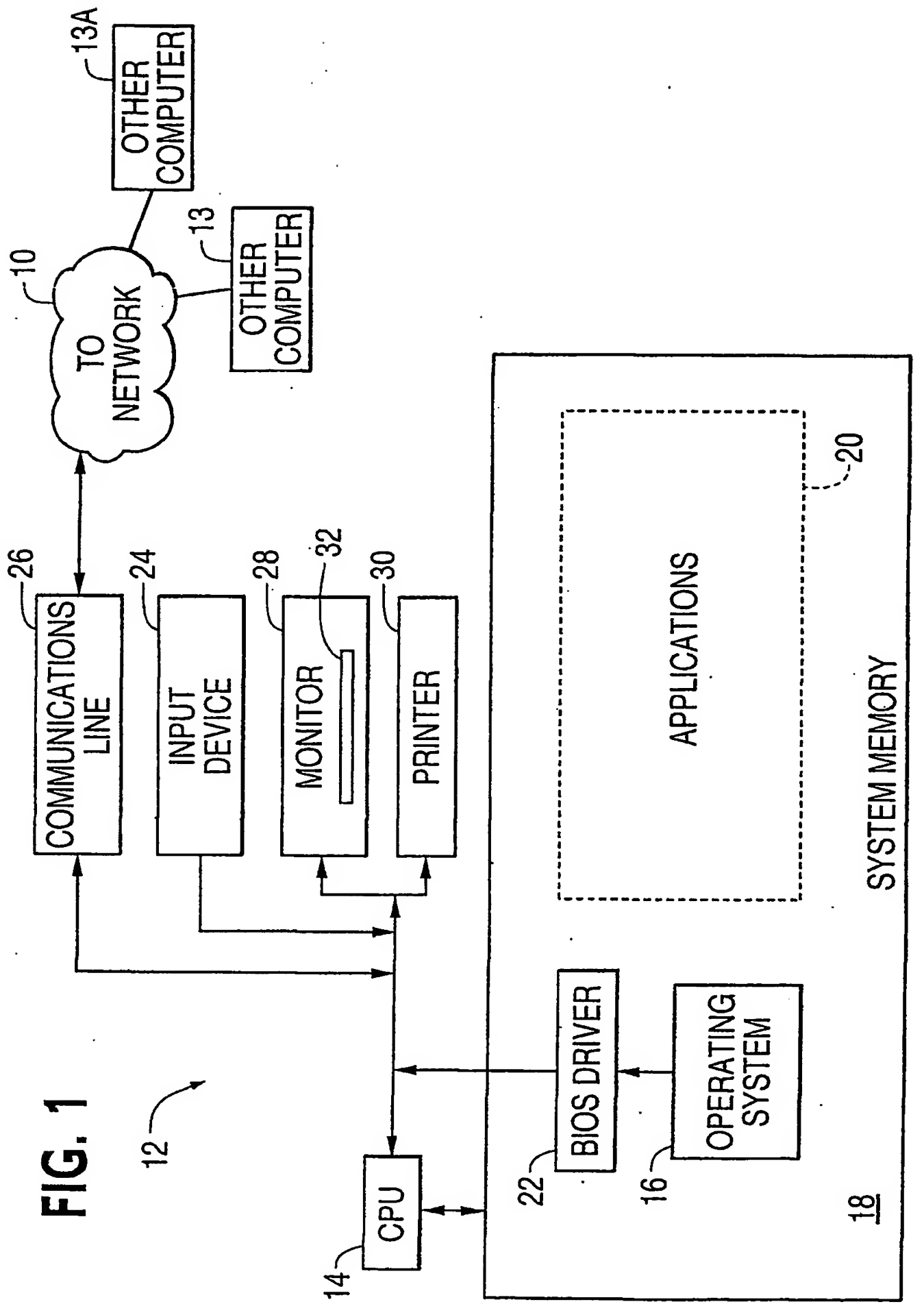

13A
OTHER COMPUTER
13
OTHER COMPUTER
10
TO NETWORK
26
COMMUNICATIONS LINE
24
INPUT DEVICE
28
MONITOR
32
30
PRINTER
20
APPLICATIONS
SYSTEM MEMORY
BIOS DRIVER
22
OPERATING SYSTEM
16
18
CPU
14
12

FIG. 1

FIG. 2

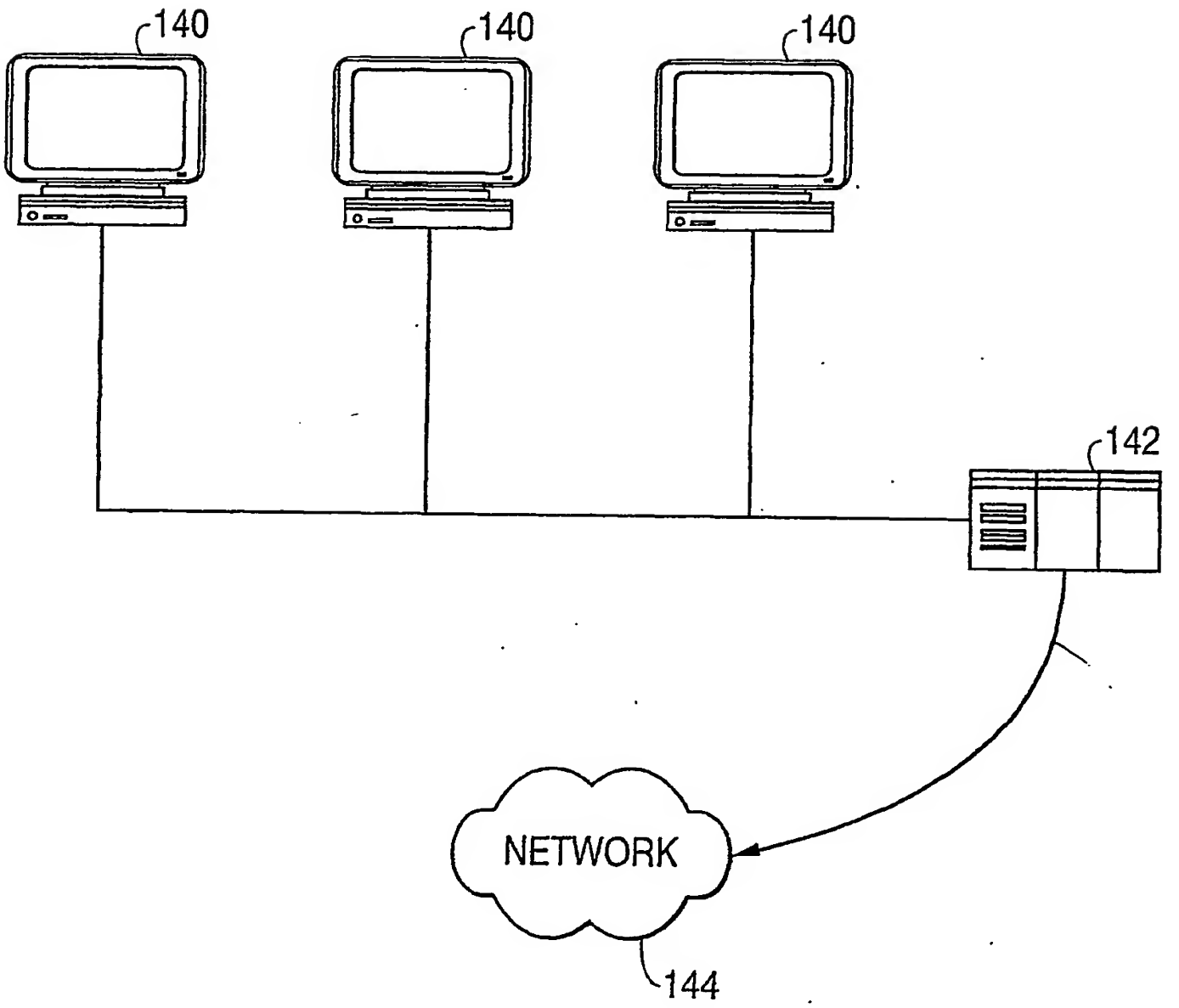
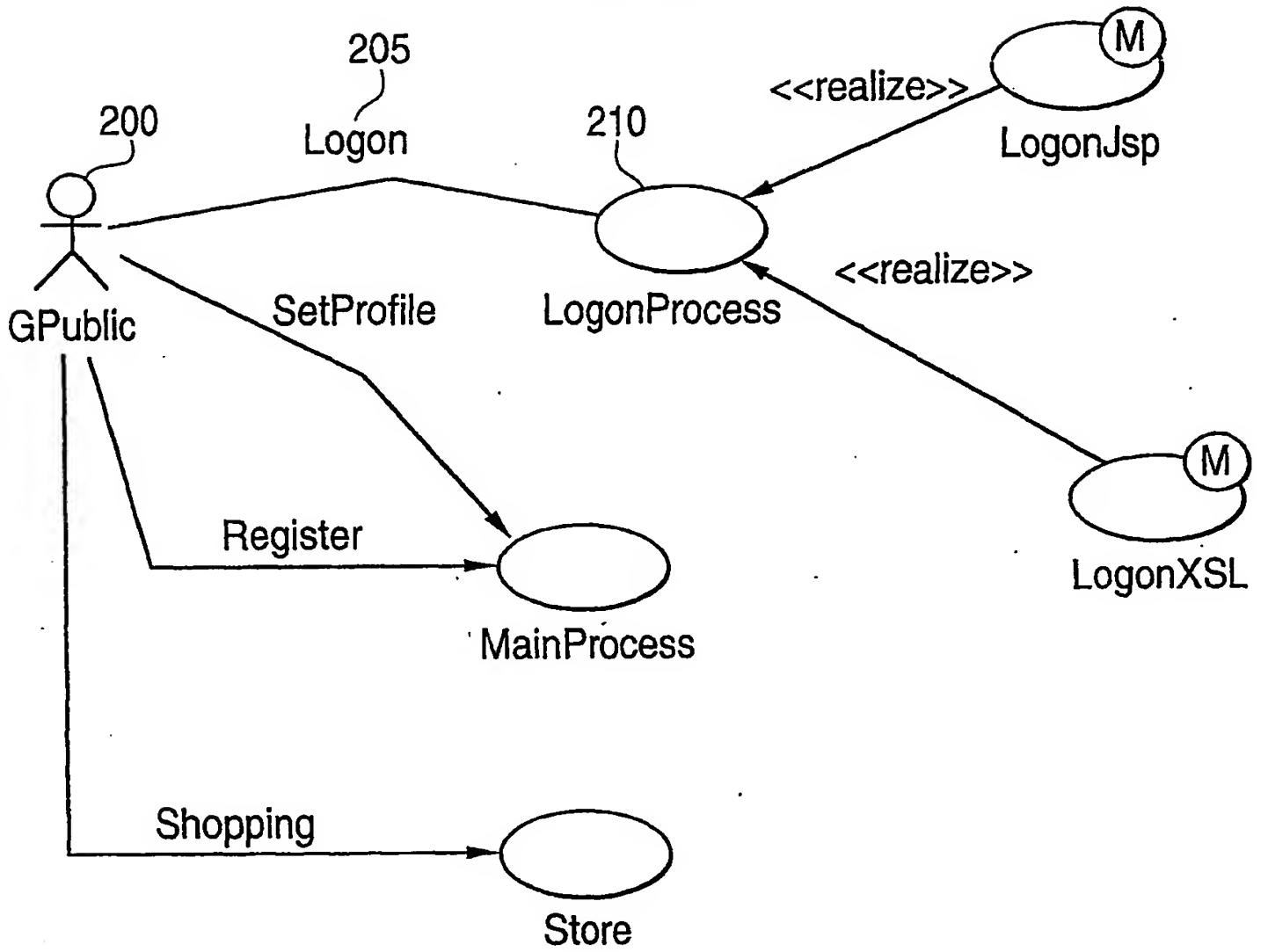


FIG. 3


205
Logon
200
210
<<realize>>
M
LogonJsp
GPublic
SetProfile
LogonProcess
<<realize>>
M
LogonXSL
Register
MainProcess
Shopping
Store

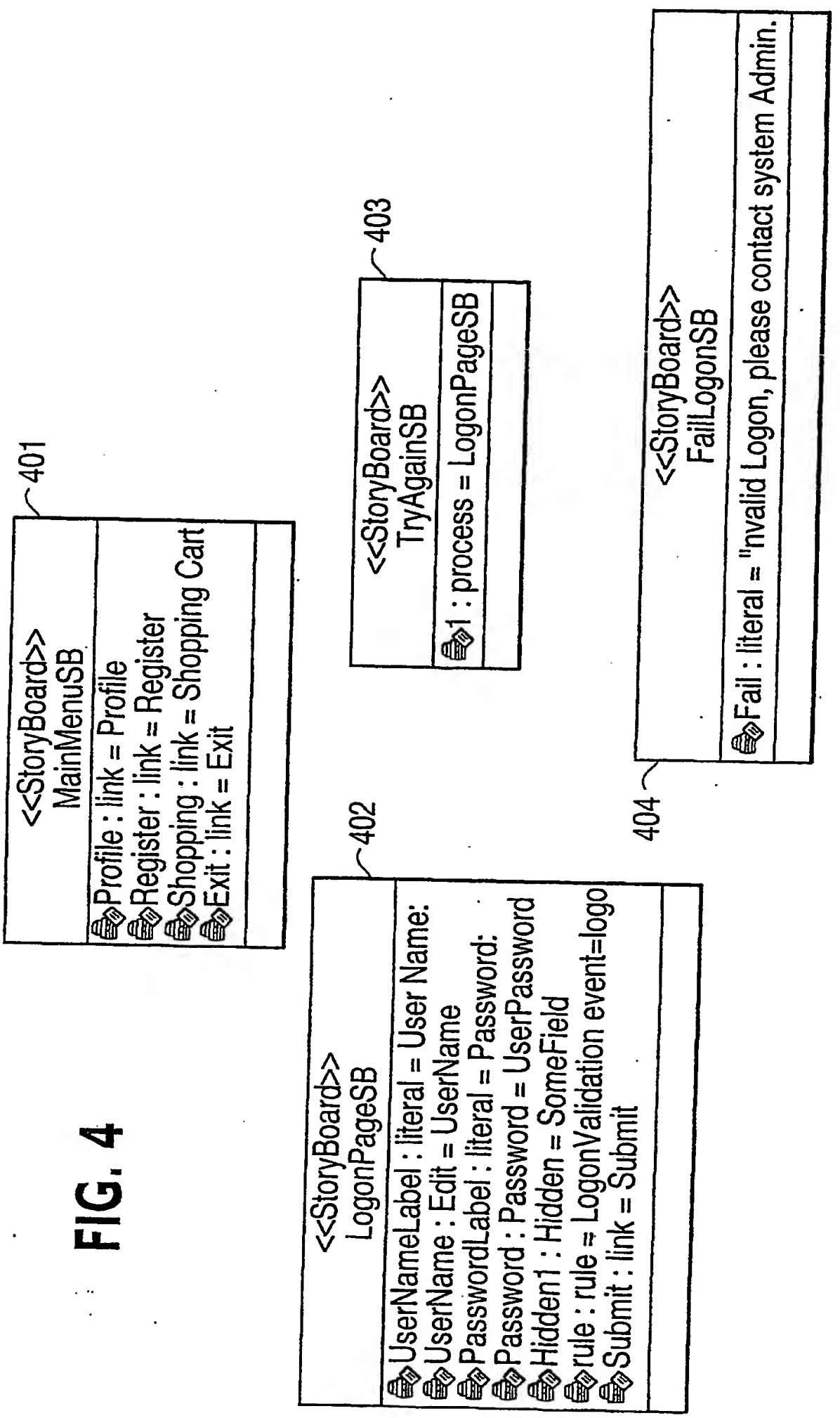

401
<<StoryBoard>>
MainMenuSB
Profile : link = Profile
Register : link = Register
Shopping : link = Shopping Cart
Exit : link = Exit
402
<<StoryBoard>>
LogonPageSB
UserNameLabel : literal = User Name:
UserName : Edit = UserName
PasswordLabel : literal = Password:
Password : Password = UserPassword
Hidden1 : Hidden = SomeField
rule : rule = LogonValidation event=logo
Submit : link = Submit
403
<<StoryBoard>>
TryAgainSB
1 : process = LogonPageSB
404
<<StoryBoard>>
FailLogonSB
eral = "nvalid Logon, please contact system Admin.
Fail : literal

FIG. 4

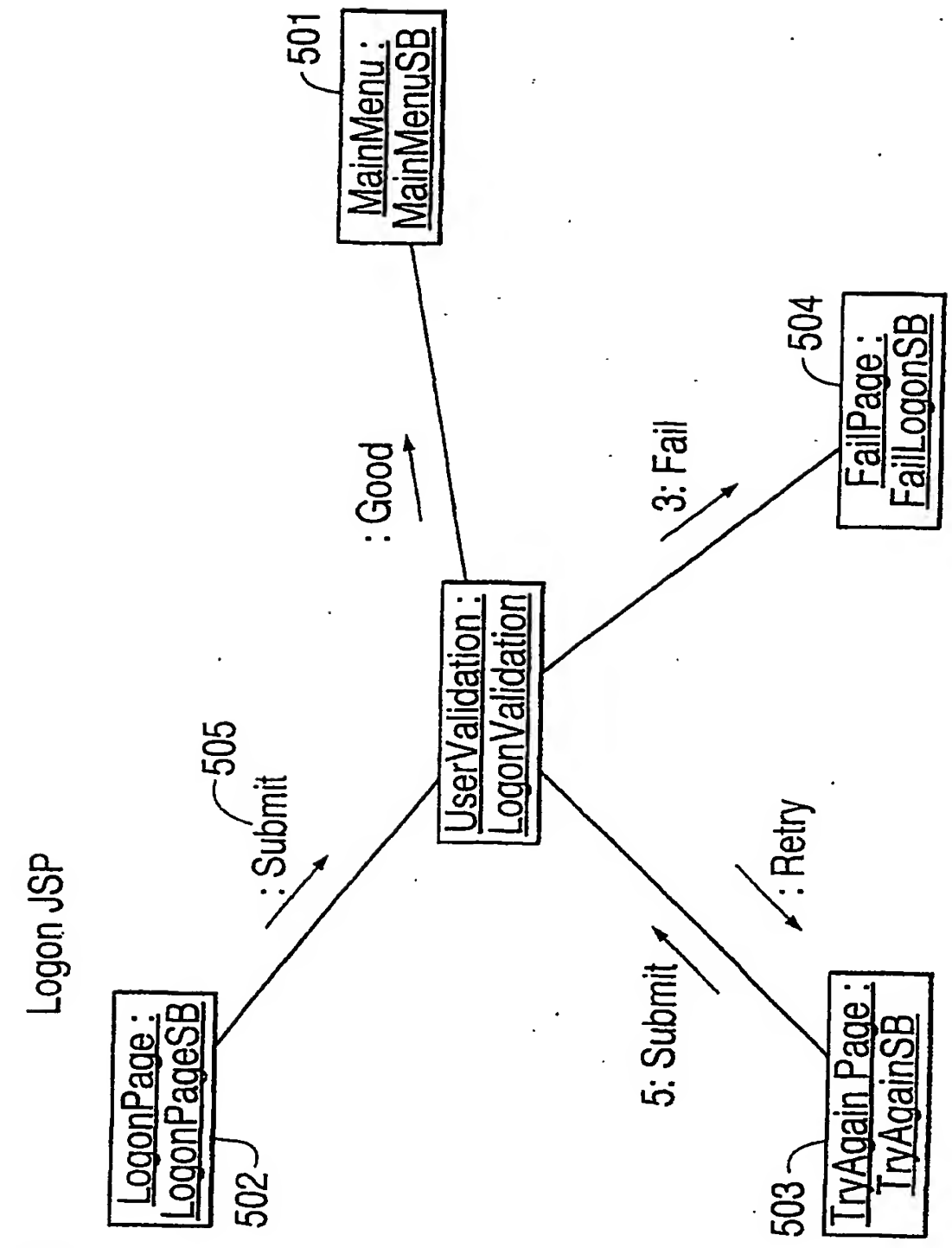

501
MainMenu : MainMenuSB
: Good
504
FailPage : FailLogonSB
3: Fail
UserValidation : LogonValidation
505
: Submit
: Retry
Logon JSP
LogonPage : LogonPageSB
502
5: Submit
503
TryAgain Page : TryAgainSB

FIG. 5

FIG. 6

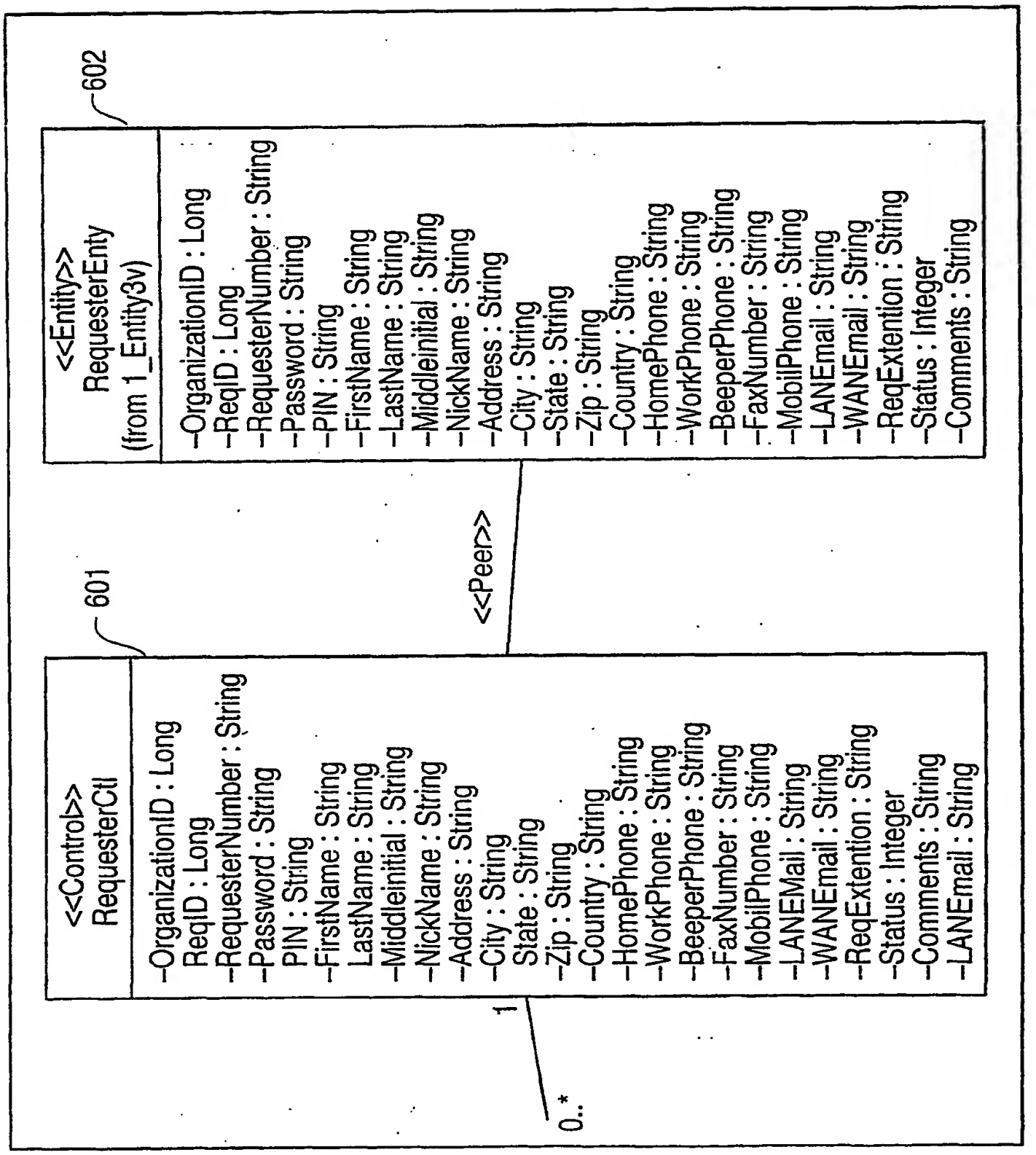

602
<<Entity>>
RequesterEnty
(from 1_Entity3v)
-OrganizationID : Long
-ReqID : Long
-RequesterNumber : String
-Password : String
-PIN : String
-FirstName : String
-LastName : String
-Middleinitial : String
-NickName : String
-Address : String
-City : String
-State : String
-Zip : String
-Country : String
-HomePhone : String
-WorkPhone : String
-BeeperPhone : String
-FaxNumber : String
-MobilPhone : String
-LANEmail : String
-WANEmail : String
-ReqExtention : String
-Status : Integer
-Comments : String
<<Peer>>
1
0..*
601
<<Control>>
RequesterCtl
-OrganizationID : Long
ReqID : Long
-RequesterNumber : String
-Password : String
PIN : String
-FirstName : String
LastName : String
-Middleinitial : String
-NickName : String
-Address : String
-City : String
State : String
-Zip : String
-Country : String
-HomePhone : String
-WorkPhone : String
-BeeperPhone : String
-FaxNumber : String
-MobilPhone : String
-LANEMail : String
-WANEmail : String
-ReqExtention : String
-Status : Integer
-Comments : String
-LANEmail : String

FIG. 7

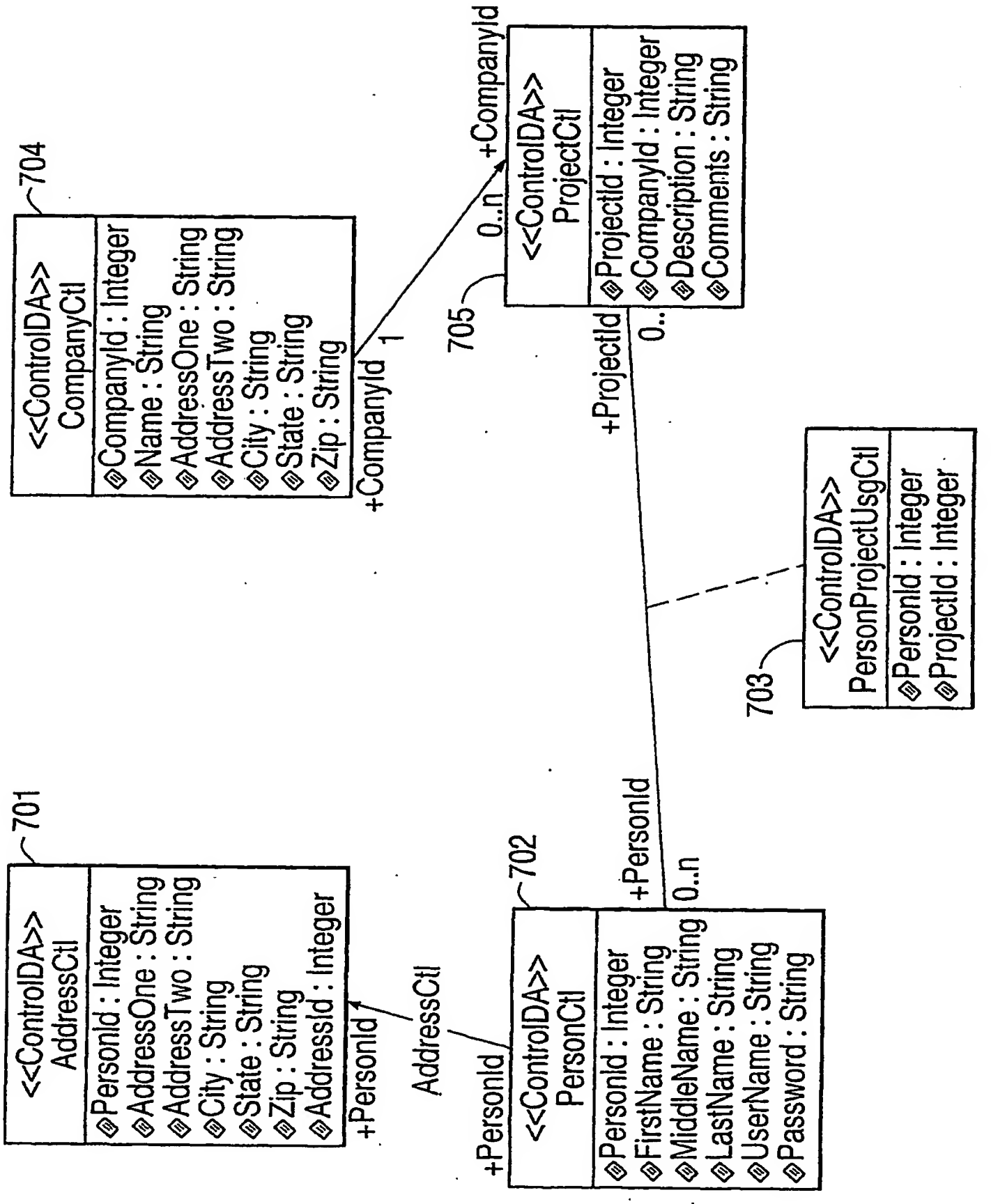

704
<<ControlDA>>
CompanyCtl
CompanyId : Integer
Name : String
AddressOne : String
AddressTwo : String
City : String
State : String
Zip : String
+CompanyId
1
0..n
+CompanyId
705
<<ControlDA>>
ProjectCtl
ProjectId : Integer
CompanyId : Integer
Description : String
Comments : String
+ProjectId
0..
703
<<ControlDA>>
PersonProjectUsgCtl
PersonId : Integer
ProjectId : Integer
701
<<ControlDA>>
AddressCtl
PersonId : Integer
AddressOne : String
AddressTwo : String
City : String
State : String
Zip : String
AddressId : Integer
+PersonId
AddressCtl
+PersonId
702
<<ControlDA>>
PersonCtl
PersonId : Integer
FirstName : String
MiddleName : String
LastName : String
UserName : String
Password : String
+PersonId
0..n

FIG. 8

```
public class PersonEnty implements Loadable, Cloneable{
                                                    // Loadable interface provides access for DBPackage
    DBPackage mDBPackage=null;                      // the Database connection this object belongs to
    DBControl mDBcontrol=null;                      // the Control this Entity belongs to
    static DBTableDefinition mDBTableDefinition=null;                    // Static field definitions
    boolean   DBPackageConnectionActive=false;  // this flag determines the access is coming from the DBPackage


    /****** Start Loadable interface ************/
    public DBTableDefinition getFields(){
       // this Loadable interface provides the DBPackage with the field definitions
       // required for object and field loading
       if (mDBTableDefinition ! = null){
          return mDBTableDefinition ;
       }
       DBTableDefinition tdef = new DBTableDefinition();
       mDBTableDefinition = tdef;
       tdef.setTableName(mTablePersonEnty);   // Database table name
       tdef.addField("PersonId", mfldPersonId.  DBPackage.dbp_Integer, 0, DBPackage.PRIMARY | DBPackage.UNIQUE |
       tdef.addField("FirstName", mfldFirstName, DBPackage.dbp_String,   50,   0);
       tdef.addField("MiddleName", mfldMiddleName, DBPackage.dbp_String,   50,   0);
       tdef.addField("LastName", mfldLastName, DBPackage.dbp_String,   50,   0);
       tdef.addField("UserName", mfldUserName, DBPackage.dbp_String,   50,   0);
       tdef.addField("Password", mfldPassword, DBPackage.dbp_String,   50,   0);
       tdef.setTimeOut( 0);
       RemoteEngineConfig rc = new RemoteEngineConfig();
       long irefresh = rc.get RemoteEngineCacheTimeOut("PersonEnty");
       if (irefresh>0){
          tdef.setTimeOut(irefresh);
       } // if(tdef.getTimeOut()>0){
       tdef.setCacheMe(false);
       return tdef;
    }
```

FIG. 9

```
public DBSchema getSchema(){
    if (mDBSchema ! = null){
      return mDBSchema;
    }
    mDBSchema = new DBSchema();
    mDBSchema.setControlName("PersonCt1");
    AssociatedControl assoc =null;
    RelationField rf=null;

    assoc = new AssociatedControl();
    assoc.setName("AddressCt1");
    assoc.setCardinality( new Integer(0));
    mDBSchema.addAssociation(assoc);
    rf = new RelationField();
    rf.setName("PersonId");
    assoc.addFields(rf);

    assoc = new AssociatedControl();
    assoc.setName("PersonProjectUsgCt1");
    assoc.setCardinality( new Integer(1));
    mDBSchema.addAssociation(assoc);
    rf = new RelationField();
    rf.setName("PersonId");
    assoc.addFields(rf);

    return mDBSchema;
}
```

FIG. 10a

```
OrganizationCtl aOrganizationCtl = new OrganizationCtl();
aOrganizationCtl.setOrgID(aRequesterCtl.getOrganizationID());
GroupingItemCtl aGroupingItemsUsageCtl1 = new GroupingItemCtl();
GroupingItemCtl aGroupingItemsUsageCtl2 = new GroupingItemCtl();
aGroupingItemsUsageCtl1.setObjectType(new Double(1));

aGroupingItemsUsageCtl1.setObjectID(aRequesterCtl.getOrganizationID());
aGroupingItemsUsageCtl2.setObjectType(new Double(2));

Vector vRelaitonship = new Vector();
vRelaitonship.addElement(aGroupingItemsUsageCtl1);
vRelaitonship.addElement(aOrganizationCtl);

Vector vRelaitonship2 = new Vector();
```

11/76

```
vRelaitonship2.addElement(aGroupingItemsU
vRelaitonship2.addElement(aFacillityCtl);

aFacillityCtl.addQuery(vRelaitonship);
aFacillityCtl.addQuery(vRelaitonship2);

Vector vaFacillityCtl=new Vector();
ii =aFacillityCtl.LoadSet(vaFacillityCtl);
```

FIG. 10b

FIG. 11

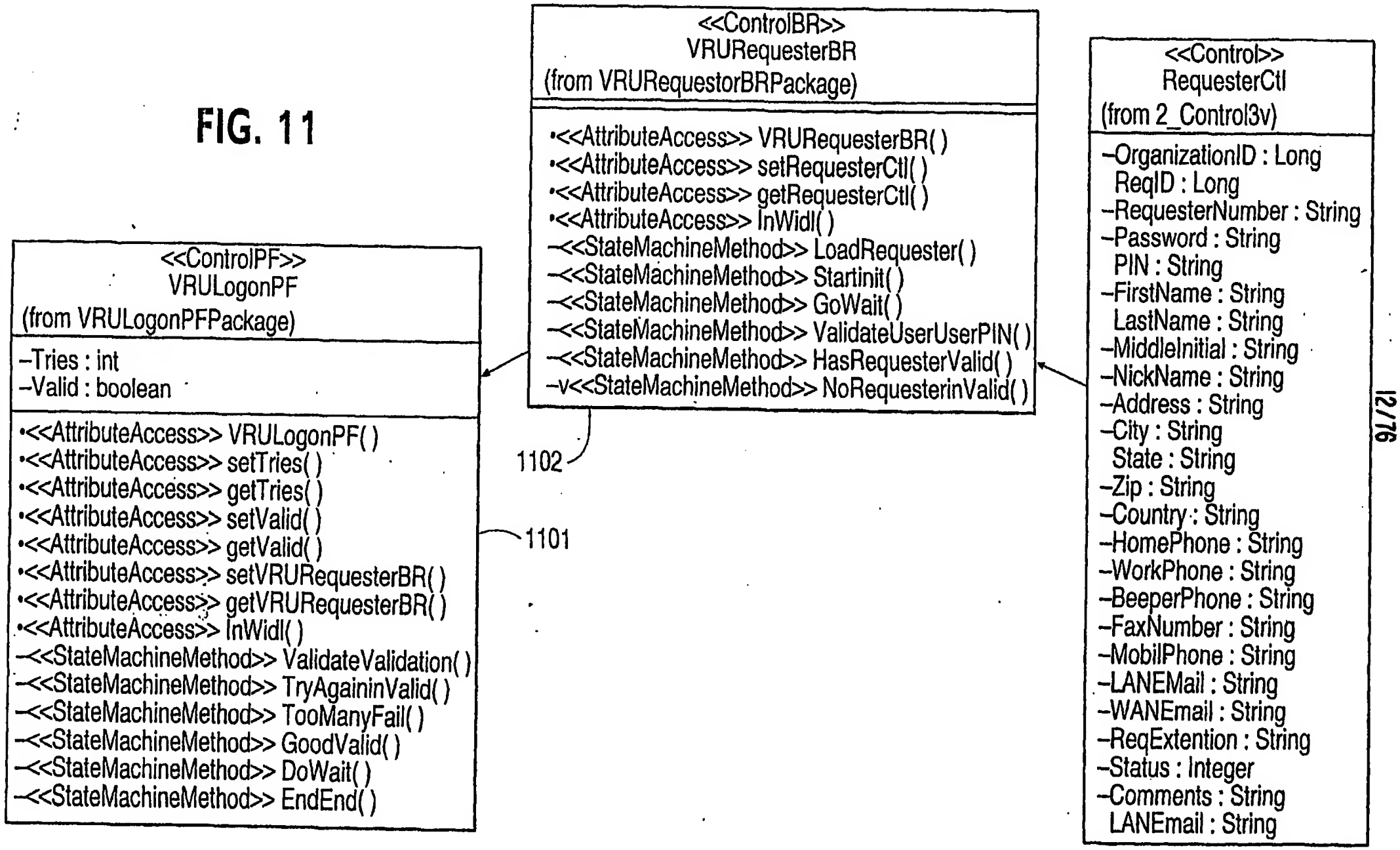


FIG. 12

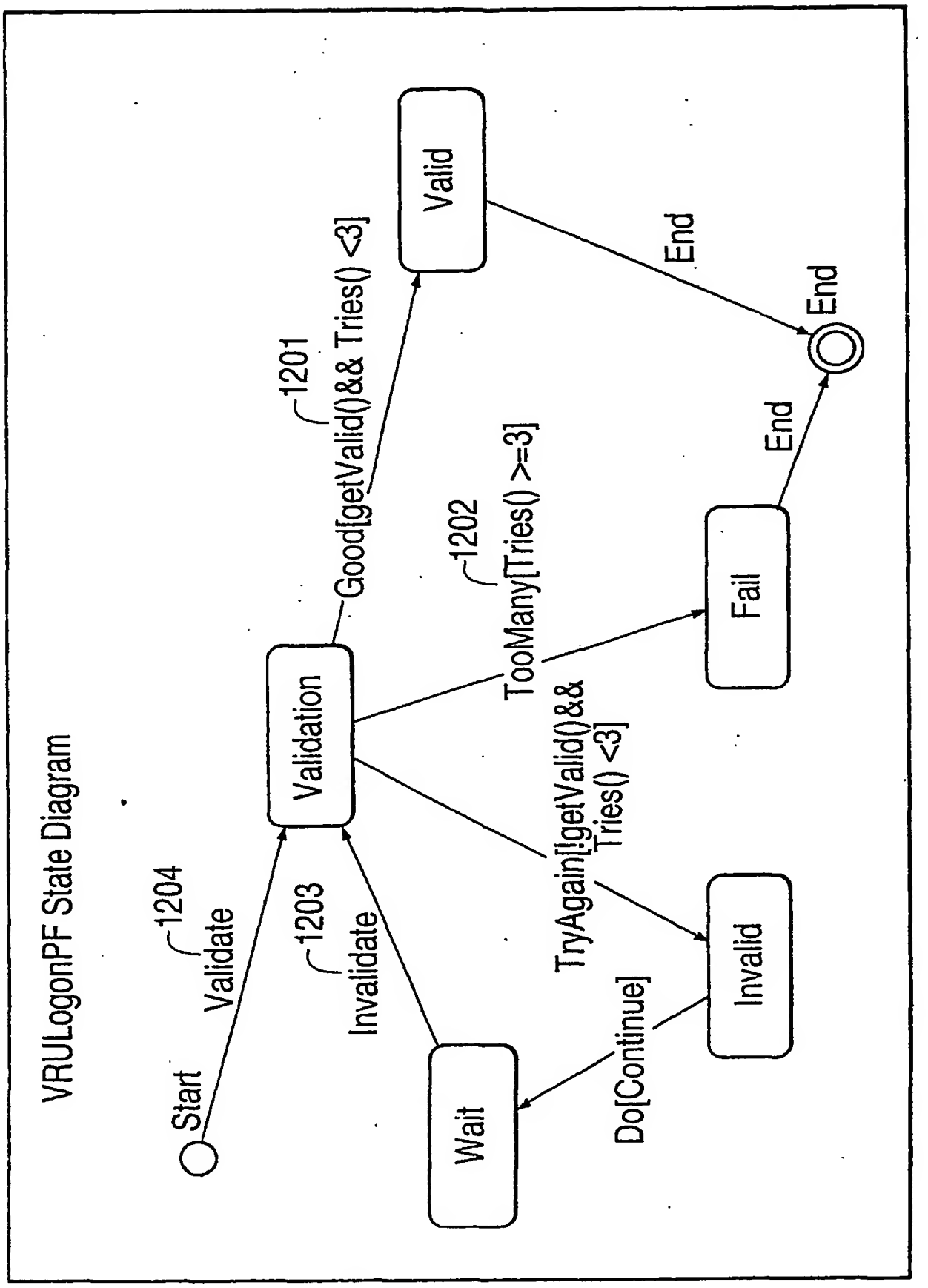

VRULogonPF State Diagram
Start
1204
Validate
Validation
1203
Invalidate
Wait
1201
Good[getValid()&& Tries() <3]
Valid
End
End
1202
TooMany[Tries() >=3]
Fail
End
TryAgain[!getValid()&& Tries() <3]
Do[Continue]
Invalid

FIG. 13a

```
/*****TWG Rational Generation*****
        // Generated StateMachine

                @roseuid -3  9
                    Regenerate Code: Yes
                *****TWG Rational Generation*****/
    public void resetStateMachine(){
        mCurrentState="Start";
    }

    public boolean inWidl(Widl theWidl){
            boolean synchronous = false;
            boolean Continue = true; // used to let the rational model read
        well on pass through states.
            boolean ret = true;
            int iState = -1;
            int iEvent = -1;
            String method ="";
```

FIG. 13b

```
do{
        method = theWidl.getMethodName();
        try {
          iState =
((Integer)mStateHash.get(mCurrentState)).intValue();
        } catch (Exception e) {iState=-1}
        try {
          iEvent = ((Integer)mEventHash.get(method)).intValue();
        } catch (Exception e) {iState=-1}
        switch(iState) {
            case isStart:
                switch (iEvent) {
                    case ieValidate:
                        mCurrentState = sValidation;
                        ValidateValidation( Widl);
                        if (!getValid()  && Tries() < 3) {
                          theWidl.setMethodName( eTryAgain);
                          synchronous = true;
                        }
                        if (Tries() >=3) {
                          theWidl.setMethodName( eTooMany);
                          synchronous = true;
                        }
                        if (getValid()  && Tries() < 3) {
                          theWidl.setMethodName( eGood);
                          synchronous = true;
                        }
                    break;

                }; // end switch (iEvent) {
            break; // isStart
            case isValidation;
                switch (iEvent) {
                    case ieTryAgain:
                        mCurrentState = sInValid;
                        TryAgainInValid( theWidl);
                        if (Continue) {
                          theWidl.setMethodName( eDo);
                          synchronous = true;
                        }
                    break;

                    case ieTooMany:
                        mCurrentState = sFail;
                        TooManyFail( theWidl);
                        synchronous = false;
                        return true;

                    case ieGood:
                        mCurrentState = sValid;
                        GoodValid( theWidl);
                        synchronous = false;
                        return true;

                }; // end switch (iEvent) {
            break; // isValidation
            case isInValid:
                switch (iEvent) {
                    case ieDo:
                        mCurrentState = sWait;
                        DoWait( theWidl);
                        synchronous = false;
```

FIG. 13c

```
                return true;

            }; // end switch  (iEvent) {
        break;  // isInValid
        case isWait:
            switch  (iEvent) {
                case ieValidate:
                    mCurrentState = sValidation;
                    ValidateValidation( theWidl);
                    if (!getValid()  && Tries() < 3) {
                        theWidl.setMethodName( eTryAgain);
                        synchronous = true;
                    }
                    if (Tries() >=3){
                        theWidl.setMethodName( eTooMany);
                        synchronous = true;
                    }
                    if (getValid()  && Tries() < 3) {
                        theWidl.setMethodName( eGood);
                        synchronous = true;
                    }
                break;

                }; //  end switch (iEvent) {
            break; //  isWait
            case isFail:
                switch (iEvent) {
                    case ieEnd:
                        mCurrentState = sEnd;
                        EndEnd( theWidl);
                        synchronous = false;
                    break;

                }; //  end switch (iEvent) {
            break; //  isFail
            case isValid:
                switch (iEvent) {
                    case ieEnd:
                        mCurrentState = sEnd;
                        EndEnd( theWidl);
                        synchronous = false;
                    break;

                }; //  end switch (iEvent) {
            break; //  isValid
        }; //  end of Switch(iState)
    } while (synchronous == true);
    return ret;
}
                    /*****TWG End @roseuid -3 ***/
```

FIG. 14

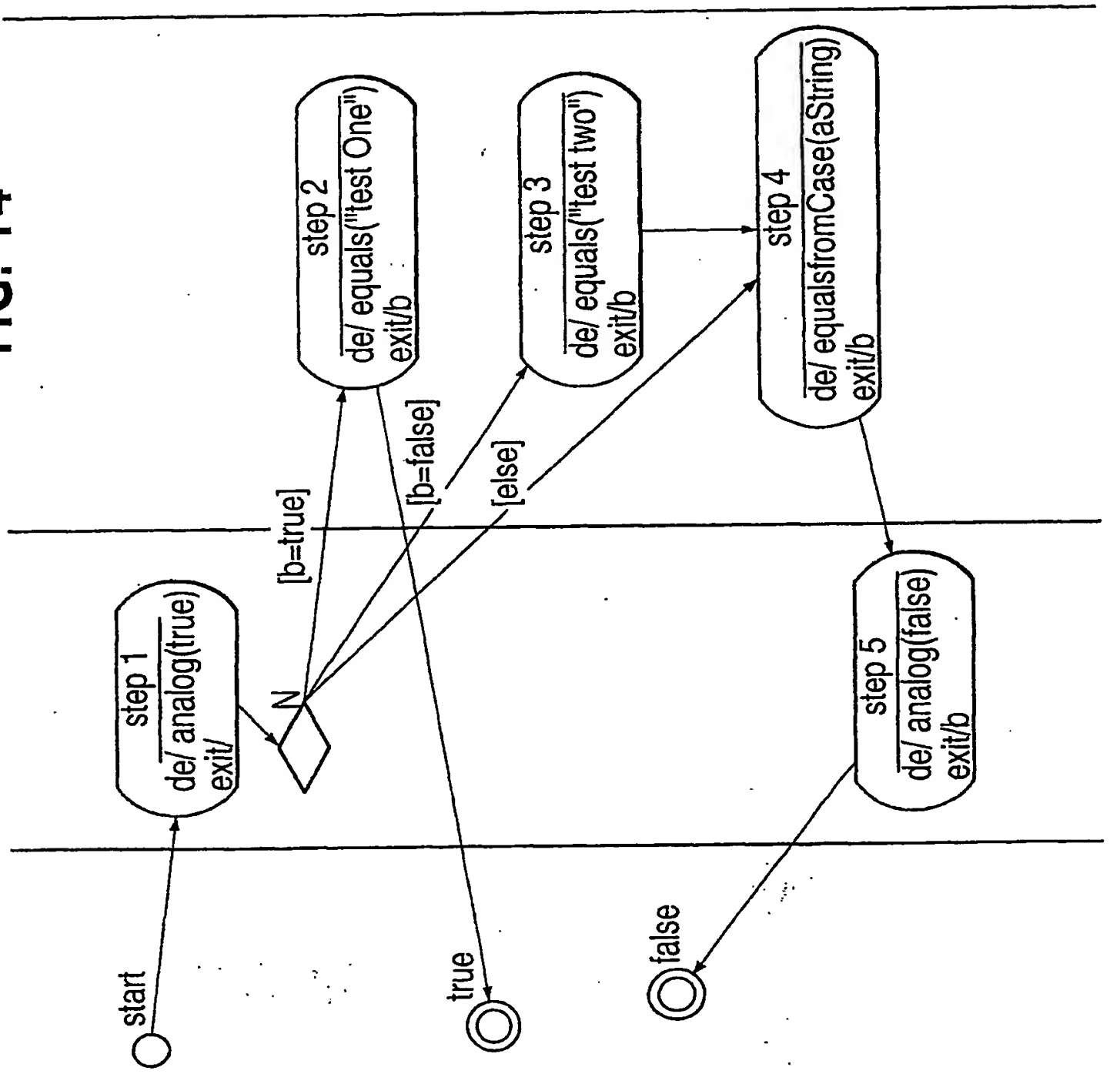

step 1
de/ analog(true)
exit/
N
[b=true]
[b=false]
[else]
step 2
de/ equals("test One")
exit/b
step 3
de/ equals("test two")
exit/b
step 4
de/ equalsfromCase(aString)
exit/b
step 5
de/ analog(false)
exit/b
start
true
false

FIG. 15

```
/*****TWG Rational Generation*****
//  <P>
// StateMachine Documentation: <P>
// Place Model notes before StateMachine
Documentation
        //  label!! <P>Event:  <P>State: Set Widl Return
result=Fail
                @roseuid 371DD79F020B 9
                Regenerate Code: Yes
        *****TWG Rational Generation*****/
    protected void TooManyFail(Widl theWidl) (
        try{
            /**
            //  <P>
            // reciever Widl theWidl:: sender VRULogonPF this
<P>
                setReturnParameter("result", "Fail")
                **/
                theWidl.setReturnParameter("result", "Fail");
            } catch (Exception e){
                System.err.println("VRULogonPF.TooManyFail: " + e);
                e.printStackTrace();
            }
        }
        /*****TWG End @roseuid 371DD79F020B *****/
```

FIG. 16

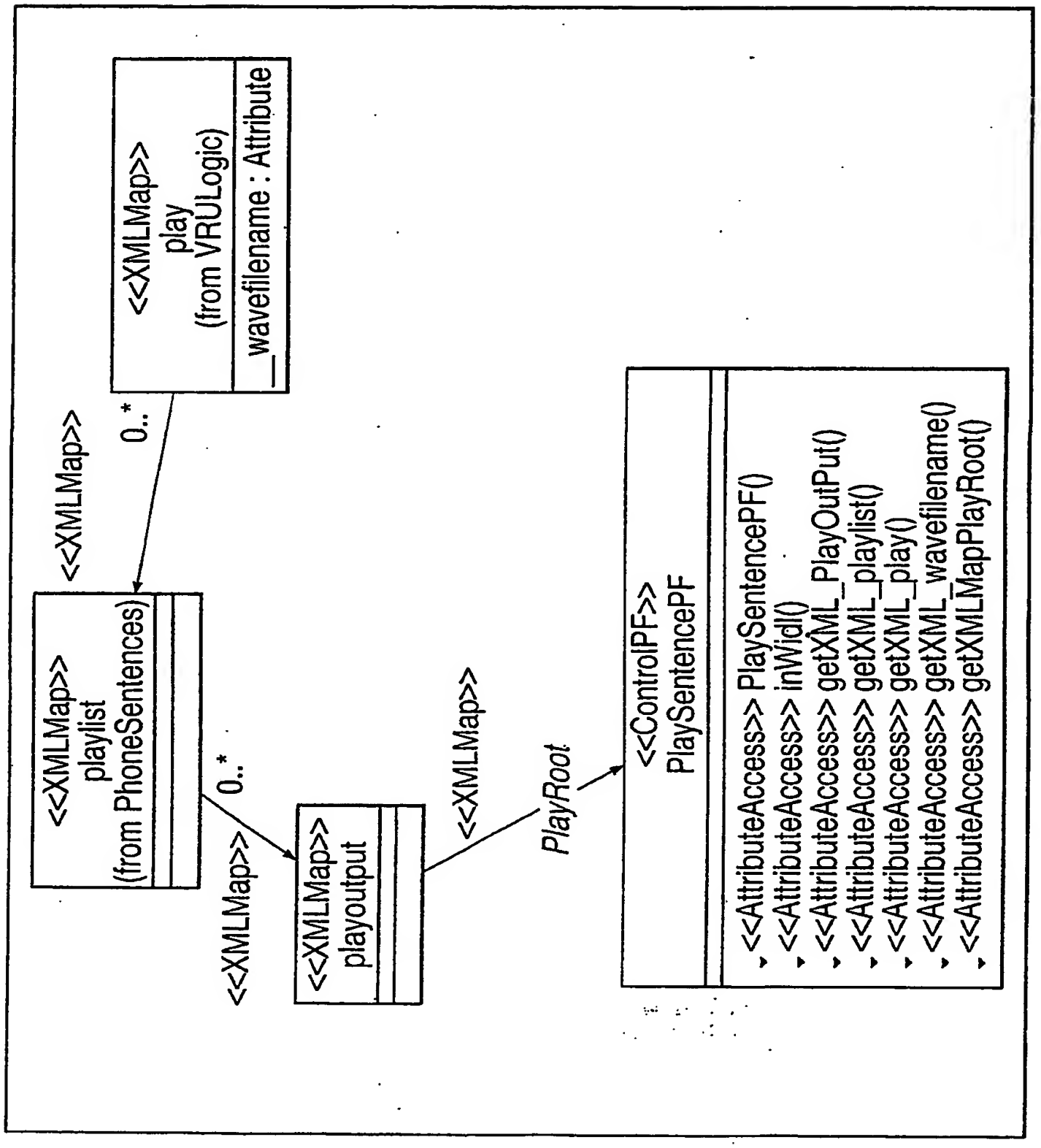

<<XMLMap>>
play
(from VRULogic)
__wavefilename : Attribute
0..*
<<XMLMap>>
<<XMLMap>>
playlist
(from PhoneSentences)
0..*
<<XMLMap>>
playoutput
<<XMLMap>>
PlayRoot
<<ControlPF>>
PlaySentencePF
<<AttributeAccess>> PlaySentencePF()
<<AttributeAccess>> inWidl()
<<AttributeAccess>> getXML_PlayOutPut()
<<AttributeAccess>> getXML_playlist()
<<AttributeAccess>> getXML_play()
<<AttributeAccess>> getXML_wavefilename()
<<AttributeAccess>> getXMLMapPlayRoot()

FIG. 17

```
/*****TWG Rational Generation*****
                                    @roseuid -4  10
                                    Regenerate Code: Yes
                                ******TWG Rational Generation*****/
                            static {
                                mXMLMapPlayRoot = new XMLMapMap();
                                mXMLMapPlayRoot.addMapping("playOutPut", "xml");
                                mXMLMapPlayRoot.addMapping("playlist", "xml* /playlist");
                                mXMLMapPlayRoot.addMapping("play", "xml/playlist* /play");
                                mXMLMapPlayRoot.addMapping("wavefilename",
                                "xml/playlist/play#wavefilename");
                            } // end static
```

FIG. 18

```
<!ELEMENT xml playlist>
<!ELEMENT playlist, play>
<!ELEMENT play>
    <!ATTLIST wavefilename VALUE CDATA >
```

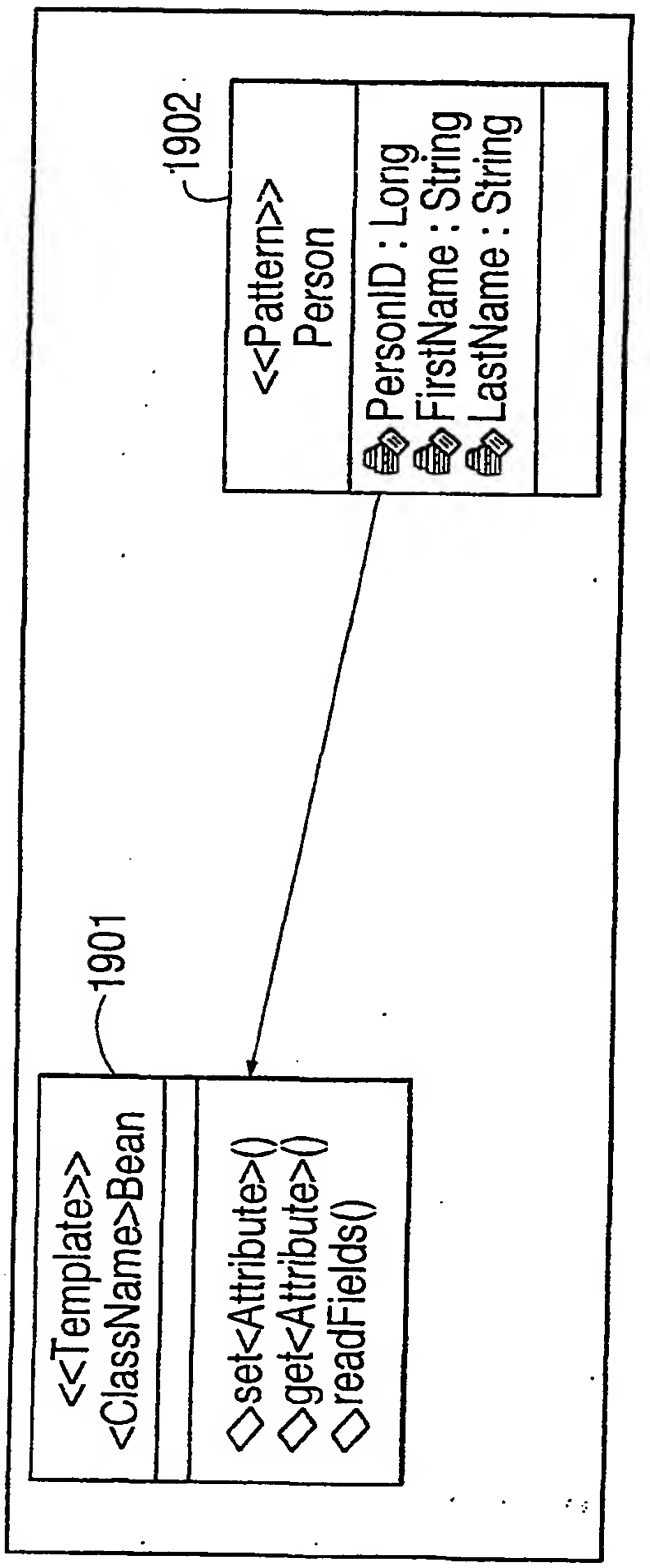

1902
<<Pattern>>
Person
PersonID : Long
FirstName : String
LastName : String
1901
<<Template>>
<ClassName>Bean
<>set<Attribute>()
<>get<Attribute>()
<>readFields()

FIG. 19

FIG. 20

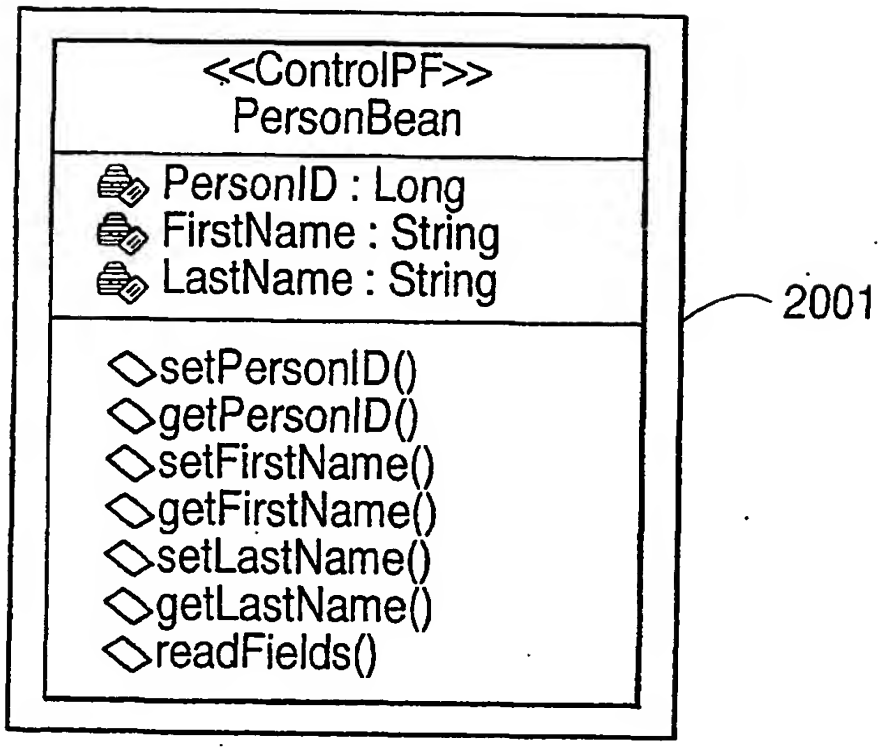

<<ControlPF>>
PersonBean
PersonID : Long
FirstName : String
LastName : String
setPersonID()
getPersonID()
setFirstName()
getFirstName()
setLastName()
getLastName()
readFields()
2001

FIG. 21

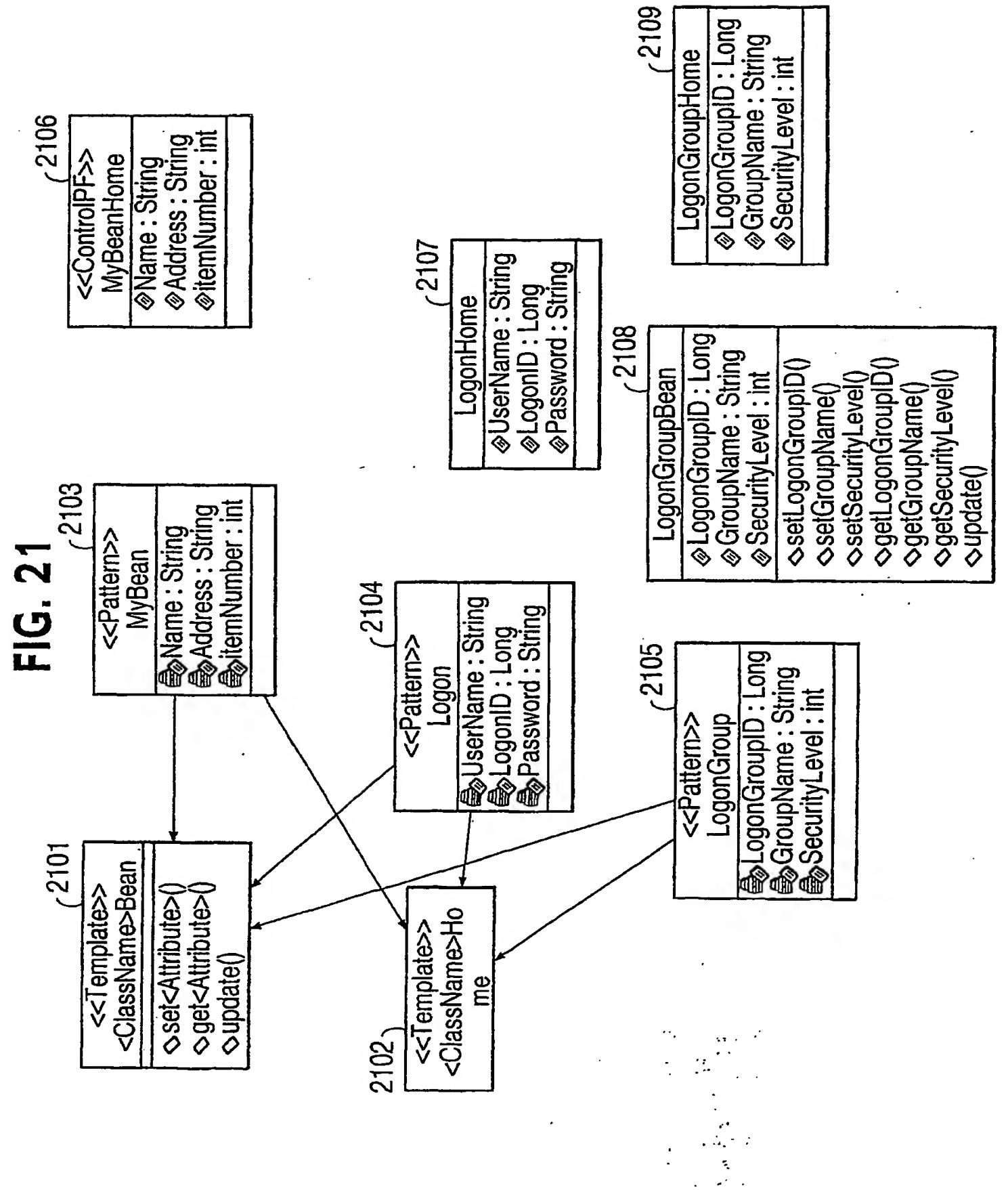
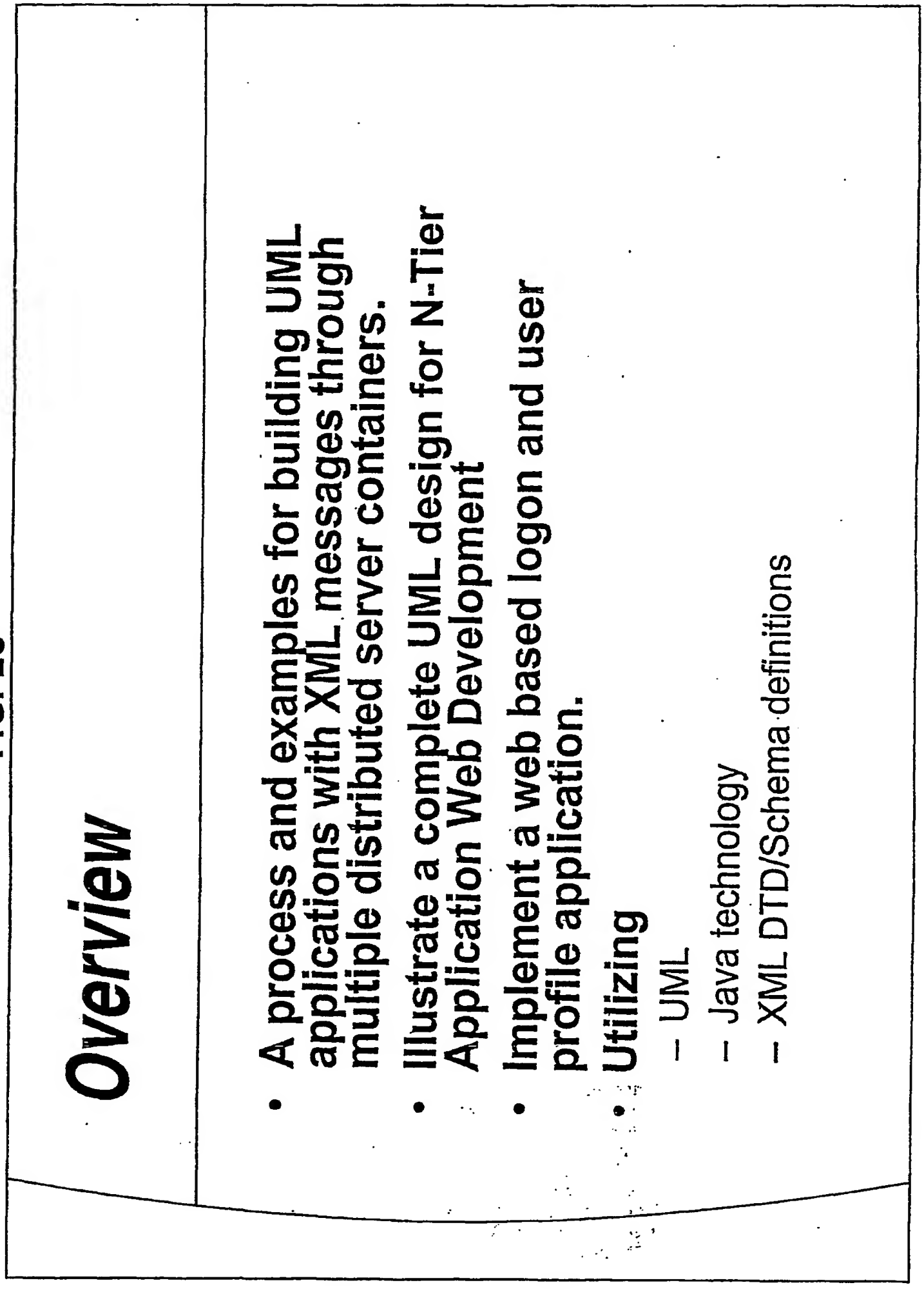

2101
<<Template>>
<ClassName>Bean
set<Attribute>()
get<Attribute>()
update()
2102
<<Template>>
<ClassName>Ho
me
2103
<<Pattern>>
MyBean
Name : String
Address : String
itemNumber : int
2104
<<Pattern>>
Logon
UserName : String
LogonID : Long
Password : String
2105
<<Pattern>>
LogonGroup
LogonGroupID : Long
GroupName : String
SecurityLevel : int
2106
<<ControlPF>>
MyBeanHome
Name : String
Address : String
itemNumber : int
2107
LogonHome
UserName : String
LogonID : Long
Password : String
2108
LogonGroupBean
LogonGroupID : Long
GroupName : String
SecurityLevel : int
setLogonGroupID()
setGroupName()
setSecurityLevel()
getLogonGroupID()
getGroupName()
getSecurityLevel()
update()
2109
LogonGroupHome
LogonGroupID : Long
GroupName : String
SecurityLevel : int

FIG. 22

Introduction

- **Effective management and modeling of enterprise applications**
- **Web and business-to-business applications**
- **Messaging environments**
  - loosely coupled
  - asynchronous
  - fault tolerant
- **Java technology provides the descriptive language**
- **XML provides the data representation.**
- **UML provides the notational language.**
- **Manage complexity for deploying n-tier enterprise applications**

FIG. 23


Overview
• A process and examples for building UML applications with XML messages through multiple distributed server containers.
• Illustrate a complete UML design for N-Tier Application Web Development
• Implement a web based logon and user profile application.
• Utilizing
– UML
– Java technology
– XML DTD/Schema definitions

Tools

- Examples presented in UML using Rational Rose® with UML Factory®.
- The examples will be implemented with JAR components that can be deployed into multiple configurations.
- UML Factory will generate, deploy, and animate the examples through the UML diagrams.

FIG. 24

FIG. 25

Technologies

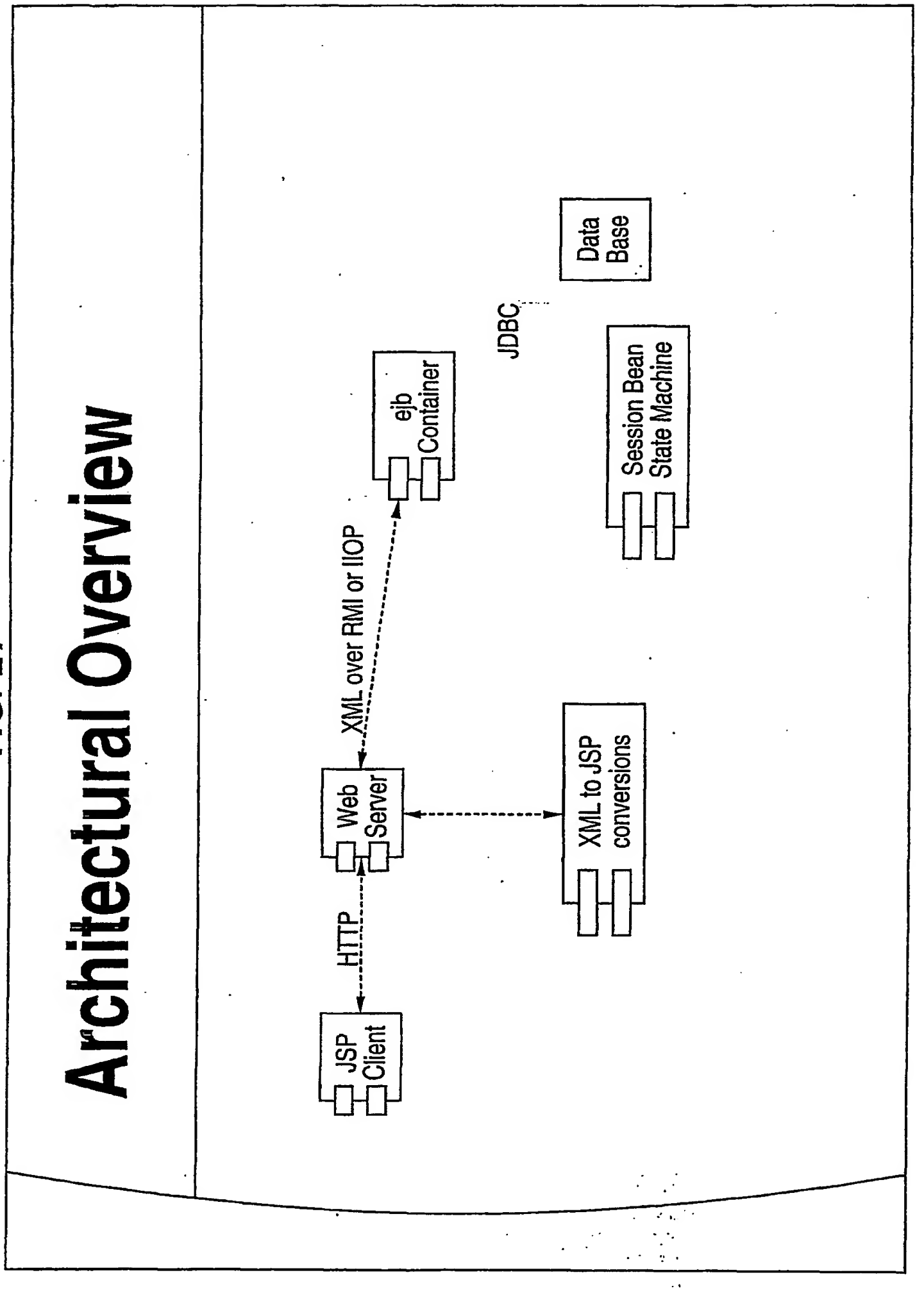
- **UML:**
  - Unified Modeling Language will describe the static and dynamic behavior for applications.
- **XML:**
  - Extensible Markup Language describes document information for building pages and carrying messages through the application tiers.
- **JSP™ components:**
  - JavaServer Pages™ components are used to define dynamic HTML interface pages

FIG. 26

Technologies

- **XSL:**
  - Extensible Style Sheet language is used to transform the XML documents into dynamic HTML interface pages.
- **EJB™**
  - Enterprise JavaBeans™ specification is used for data access and manipulation.
- **JMS:**
  - Java™ Message Service API provides an asynchronous fault tolerant messaging capability between application tiers.

FIG. 27


Architectural Overview
Data Base
JDBC
ejb Container
Session Bean State Machine
XML over RMI or IIOP
Web Server
XML to JSP conversions
HTTP
JSP Client

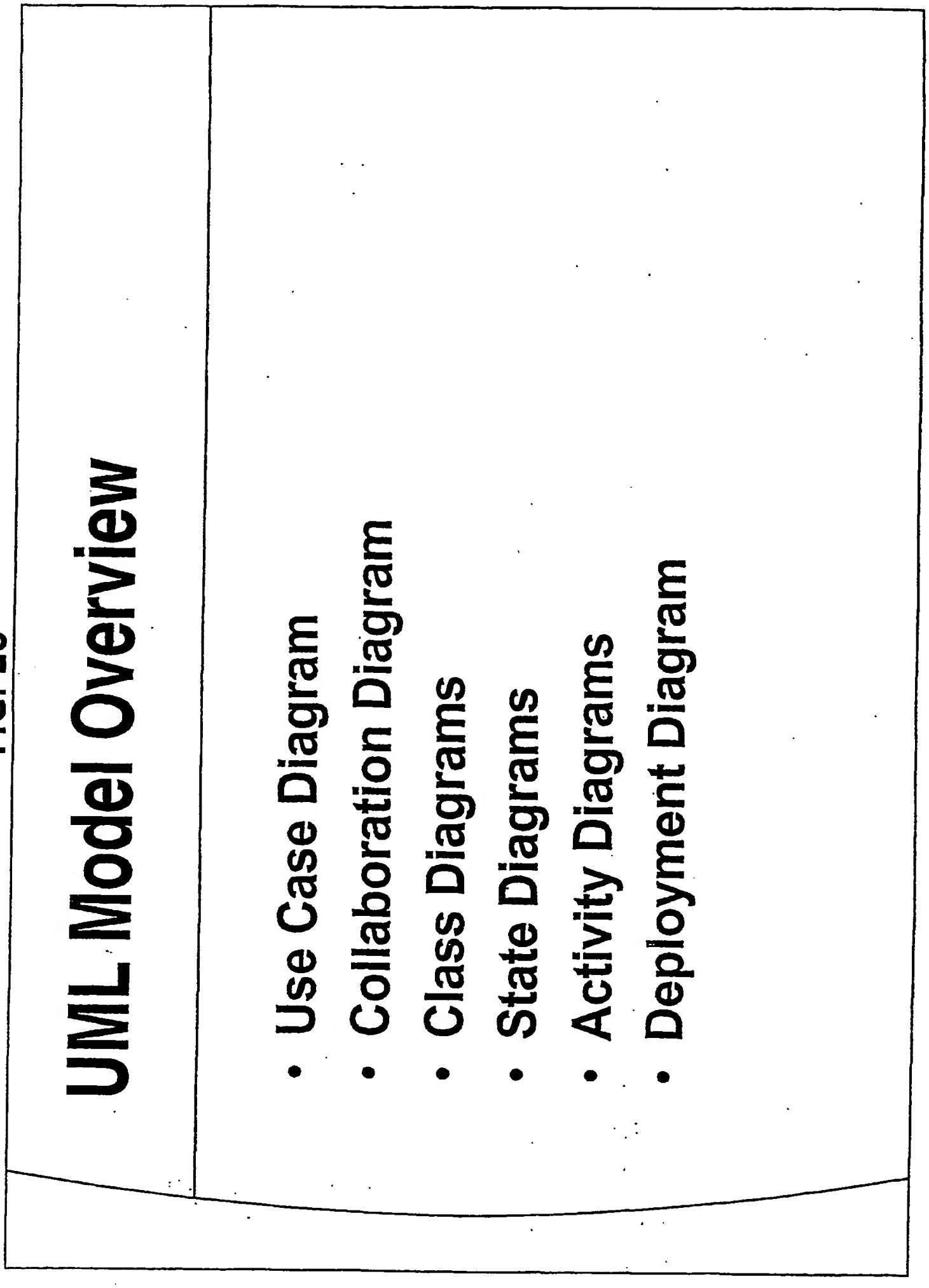

UML Model Overview
• Use Case Diagram
• Collaboration Diagram
• Class Diagrams
• State Diagrams
• Activity Diagrams
• Deployment Diagram

FIG. 28

FIG. 29

Modeling the Application

- **Use Case**
  - Use Case Diagrams define the high-level interactions between external actors and system processes. The Use Case diagram must have an actor, an interaction, and a process.

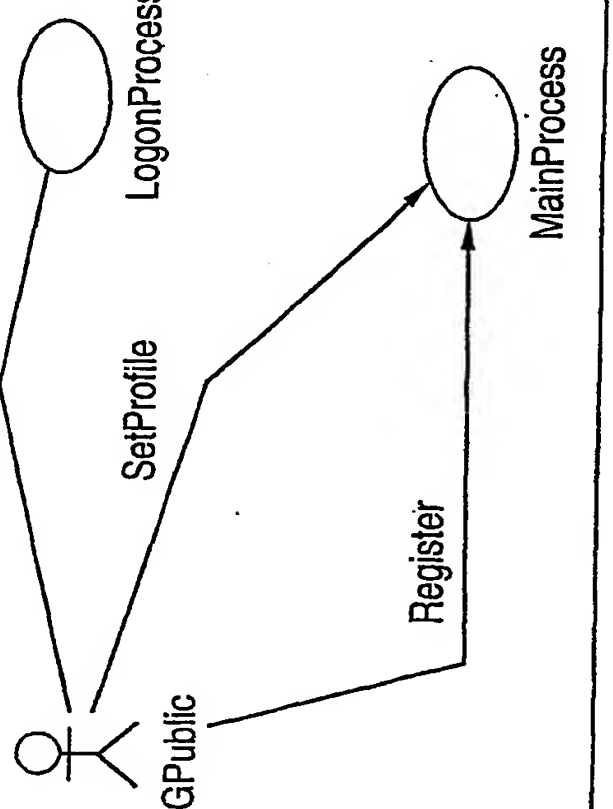


FIG. 30

Alternate Implementations

- **Realize.**
  - The "realize" stereotype on a use case interaction defines alternate mechanisms for implementing the same process.

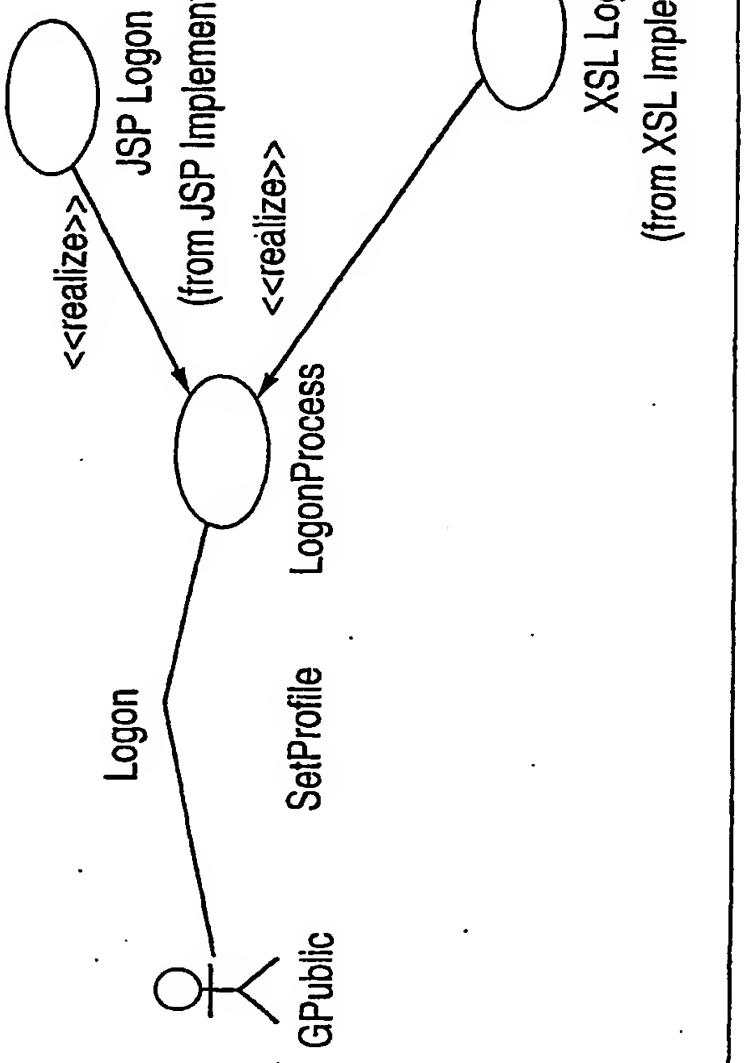


FIG. 31

User Artifacts

- Each interaction on a use case process contains artifacts, these artifacts are tangible attributes provided to the actor or input artifacts from the actor to the process. For a logon:
  - Username
  - UserPassword

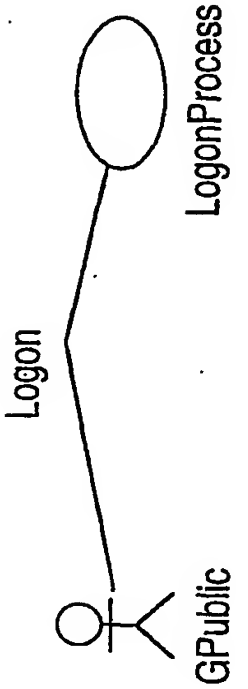


FIG. 32

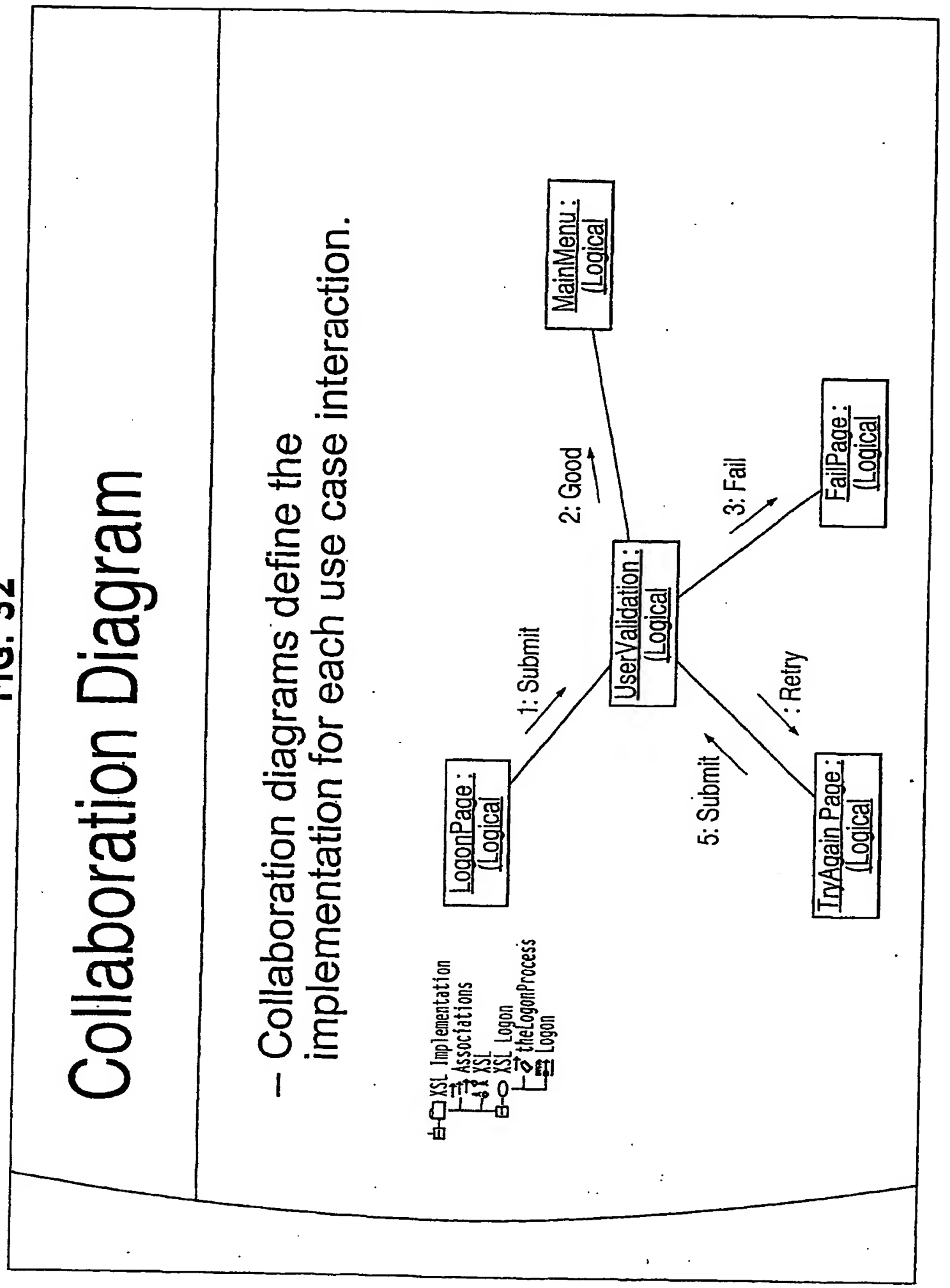

Collaboration Diagram
– Collaboration diagrams define the implementation for each use case interaction.
MainMenu : (Logical
2: Good
3: Fail
FailPage : (Logical
LogonPage : (Logical
1: Submit
UserValidation : (Logical
5: Submit
: Retry
TryAgain Page : (Logical
XSL Implementation
Associations
XSL
XSL Logon
theLogonProcess
Logon

FIG. 33

Storyboard Interface Objects

– Storyboard interface objects identify user input and output artifacts. These objects will identify the types of artifacts used within the system for the use case interaction.

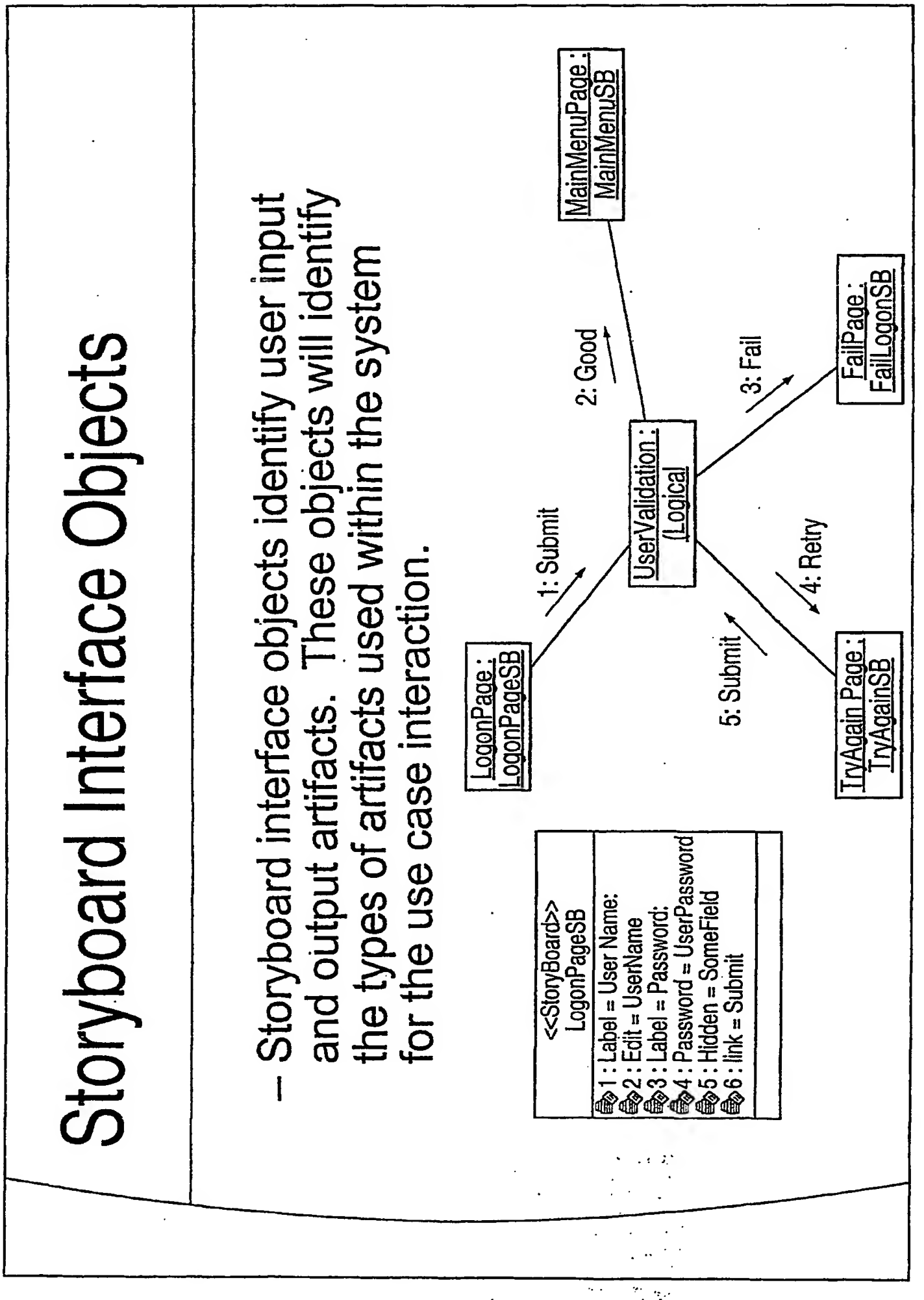


FIG. 34

Decision Control Process Flow Objects

– Through the collaboration diagram choices or decisions are made directing the diagram flow. Control process flow objects define the domain logic class specifications that will govern these decisions.

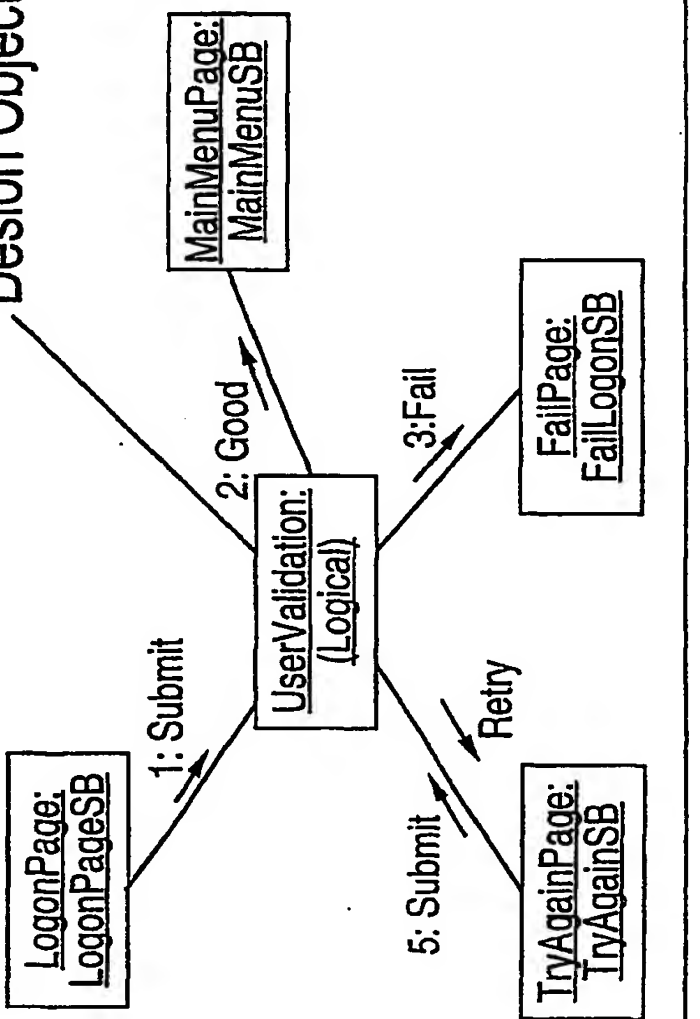


FIG. 35

Events

– Collaboration diagram events show the linking between various collaboration diagram objects.

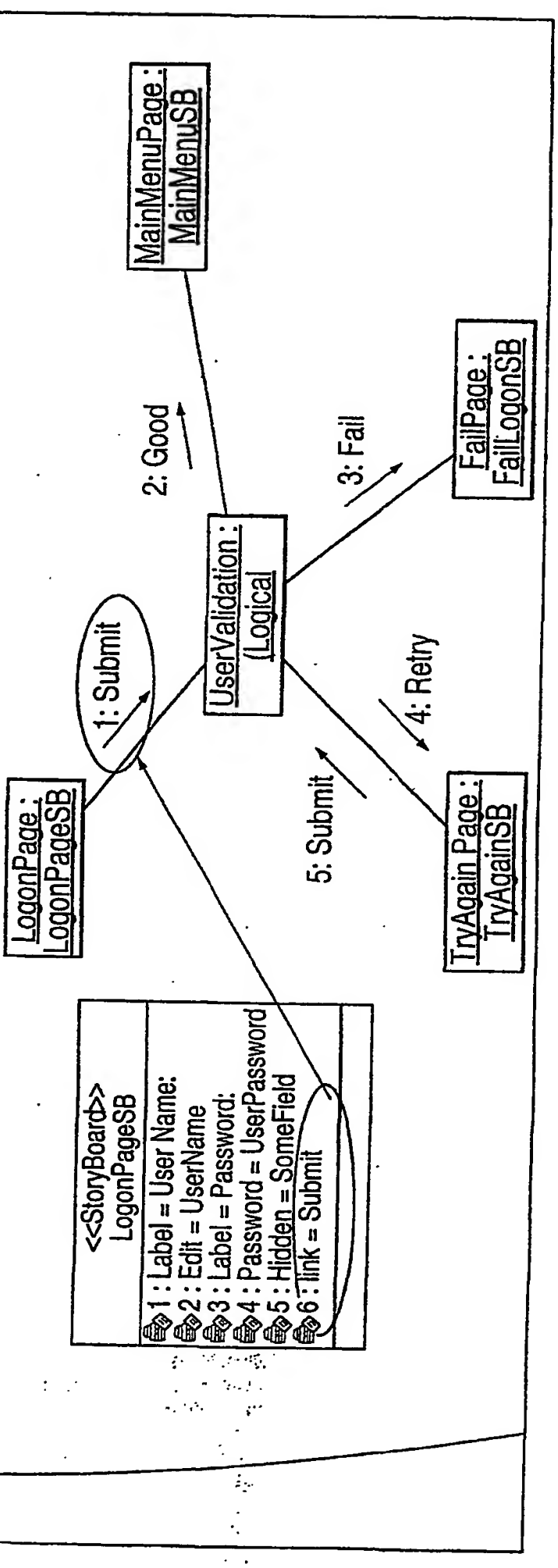


FIG. 36

Storyboard Class Specifications

– The storyboard class specification identifies the ordered set of artifacts and events that this storyboard object element will contain.

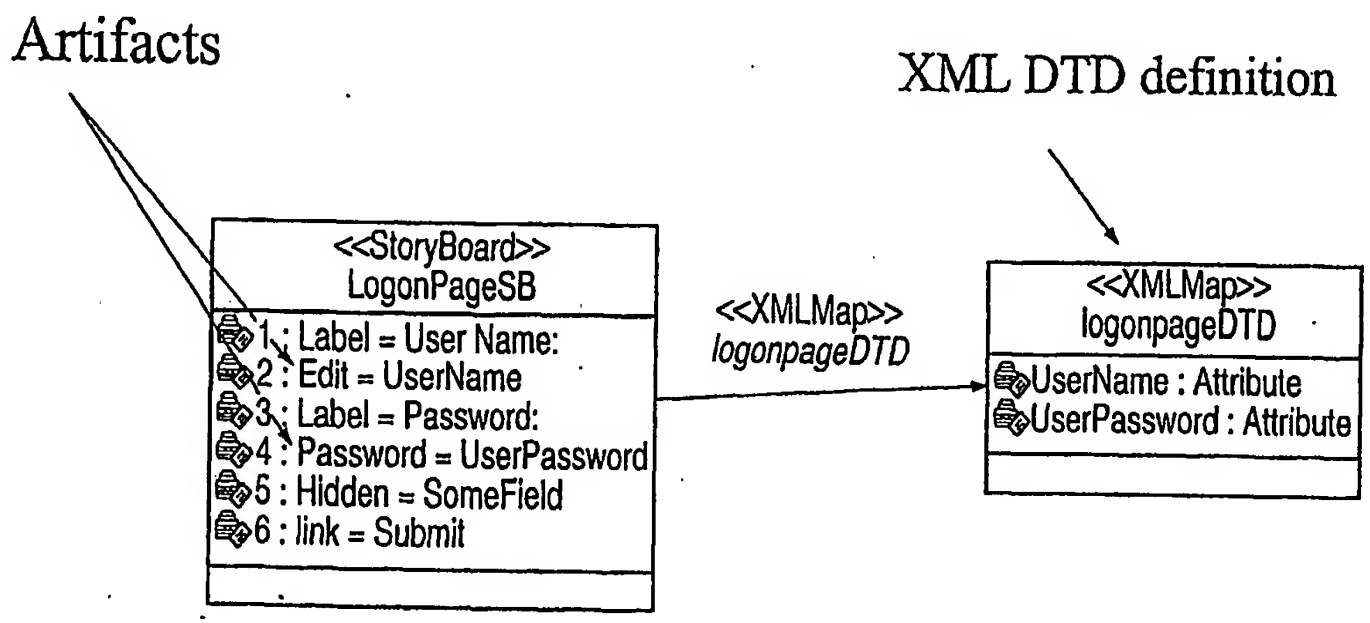


FIG. 37

HTML Pages

– Each storyboard class specification is linked to an HTML page that will be processed to create the XSL or JSP interface definition.

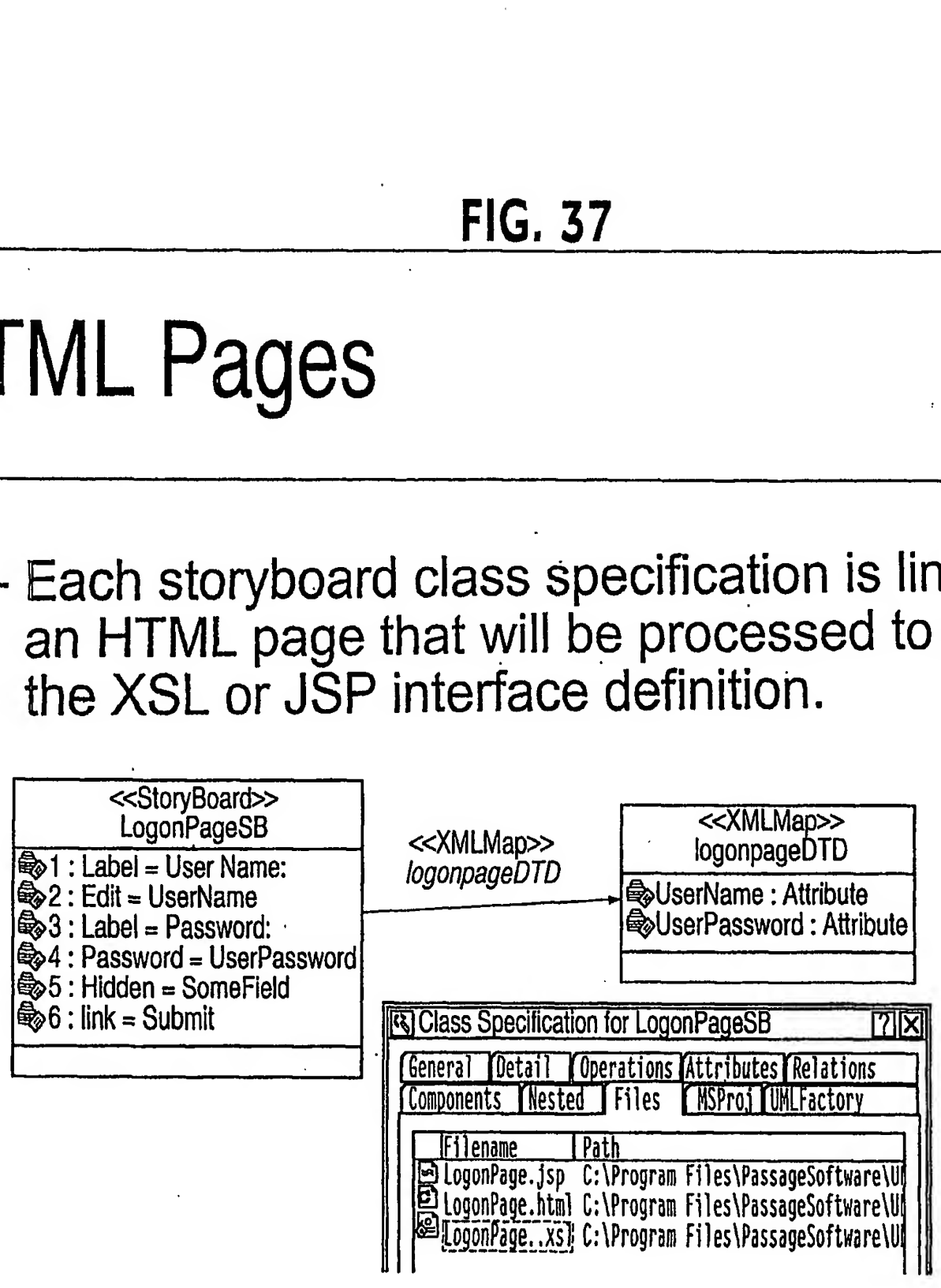


FIG. 38

HTML Pages

– The HTML pages will contain tokens identifying the XML abstract semantic names representing the use case artifacts coming and going from the control process flow.

FIG. 39

XML DTD/Schema Modeling

– Modeling the XML schema information within UML provides a visual representation of the XML documents structure. The modeled XML document also provides runtime information and model time checking for collating the use case artifacts with the XML elements.

• Sample Company Person XML

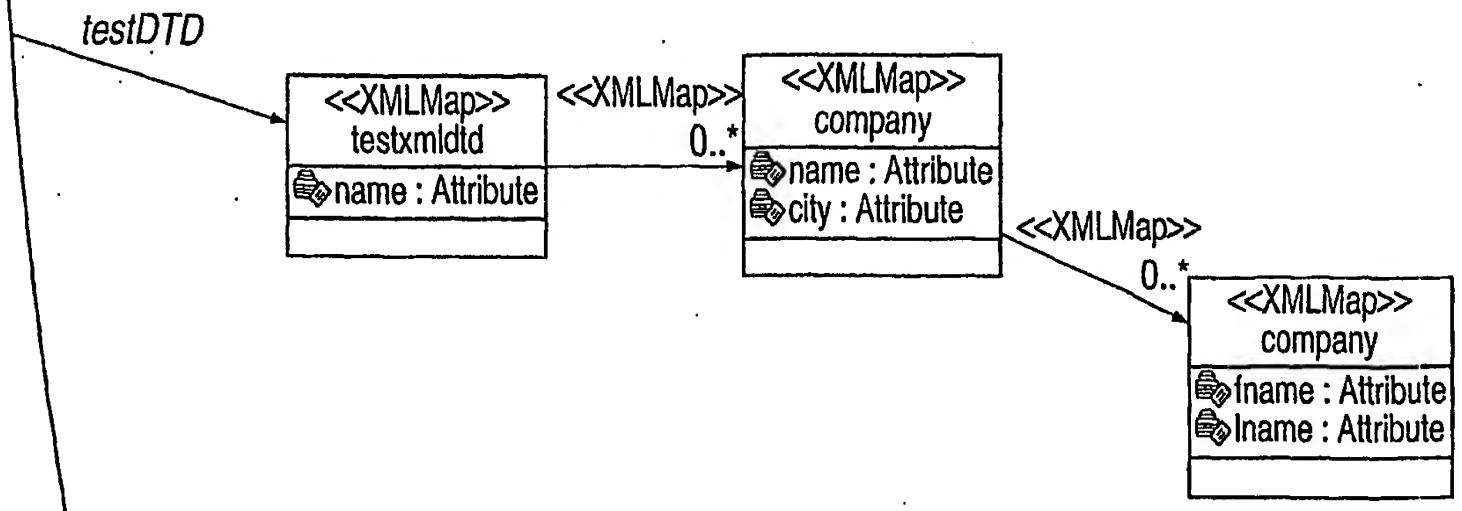


FIG. 40

XML Mappings

- **UML Factory Provides an abstract mapping between XML elements and semantic names. These Mappings allow isolation between the arbitrary physical representation of a data element, and a logical name or handle to access and manipulate that element.**
  - *Semantic Names*
    - Semantic Names Identify abstract textual identifiers for elements, or Attributes, within the XML document.
  - *XML Element Mappings*
    - XML element mappings identify the arbitrary physical XML element definition. The XML mappings identify XML text, cardinality, attributes, and schema information

FIG. 41

XML Mapping example

– Generated semantic mappings to sample Company Person XML

```
mXMLMaptestDTD = new XMLMap();
mXMLMaptestDTD.addMapping("TestXML", "xml");
mXMLMaptestDTD.addMapping("XmlName", "xml/name");
mXMLMaptestDTD.addMapping("Company", "xml /company");
mXMLMaptestDTD.addMapping("CompanyName", "xml/company#name");
mXMLMaptestDTD.addMapping("CompanyCity", "xml/company#city");
mXMLMaptestDTD.addMapping("Person", "xml/company* /person");
mXMLMaptestDTD.addMapping("FirstName", "xml/company/person#fname");
mXMLMaptestDTD.addMapping("LastName", "xml/company/person#lname");
```

FIG. 42

XML Mappings to "JSP tags"

– The token semantic names within the HTML page associate with storyboard class specifications are substituted with methods to extract semantic data from the XML document representing the Storyboard interface object.

FIG. 43

XML Mappings to XML tags

- In like manner, the token semantic names within the HTML page associated with storyboard class specifications are replaced with XSL syntax to transform the XML document representing the Storyboard interface object.

FIG. 44

Storyboard Events

- **The storyboard class specification objects contain events that will fire into the application, transitioning through the designed collaboration diagrams.**
  - If the target object of a collaboration diagram event is another interface element then that storyboard interface element will be displayed.
  - If the target object is a decision point then the class specification defined for that decision logic would be sent the event through the UML state machine implementation.

FIG. 45

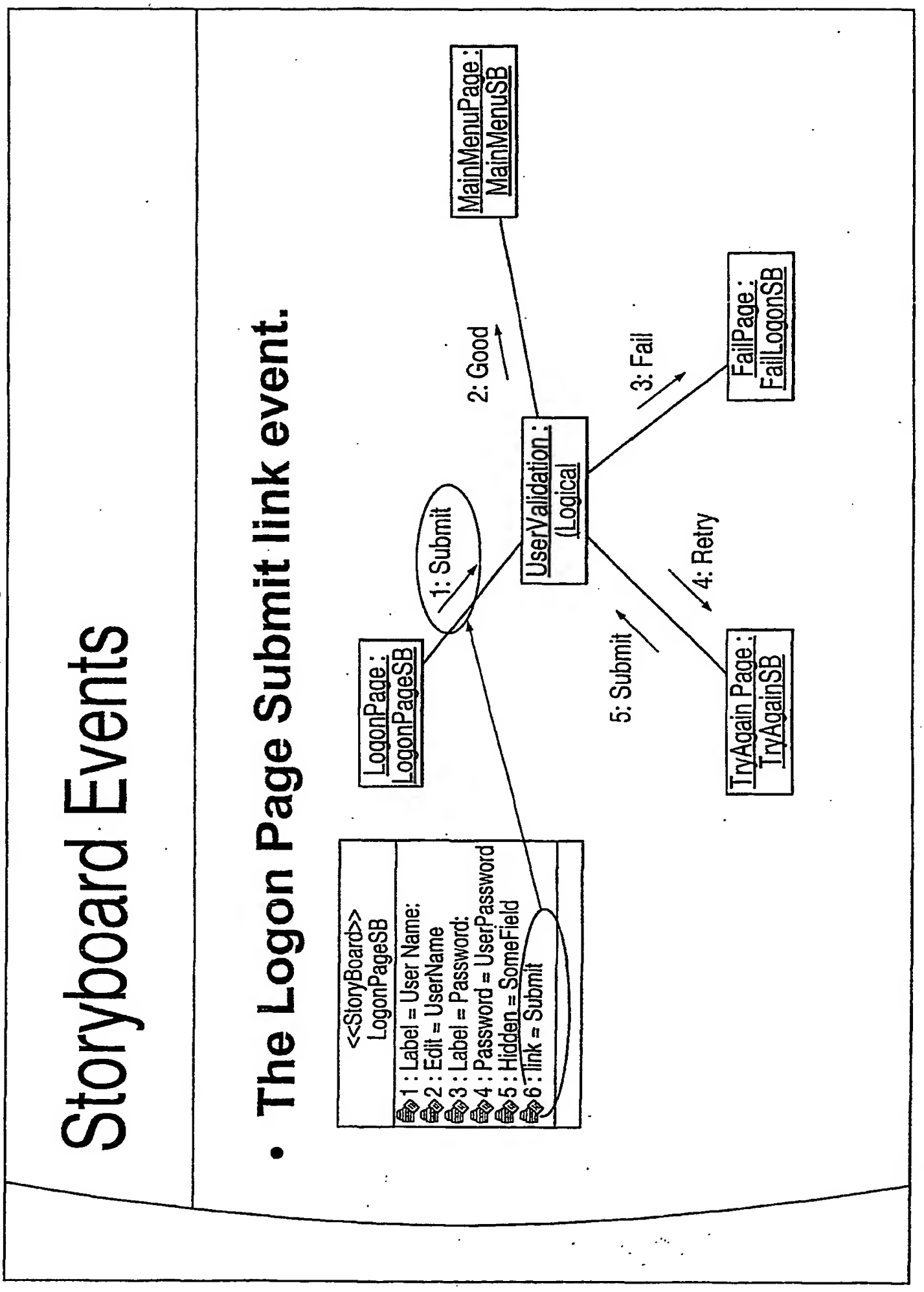

Storyboard Events
• The Logon Page Submit link event.
<<StoryBoard>>
LogonPageSB
1 : Label = User Name:
2 : Edit = UserName
3 : Label = Password:
4 : Password = UserPassword
5 : Hidden = SomeField
6 : link = Submit
LogonPage : LogonPageSB
1: Submit
2: Good
UserValidation : (Logical
3: Fail
FailPage : FailLogonSB
4: Retry
5: Submit
TryAgain Page : TryAgainSB
MainMenuPage : MainMenuSB

FIG. 46

Collaboration Decisions

– Decisions are implemented by UML logical classes

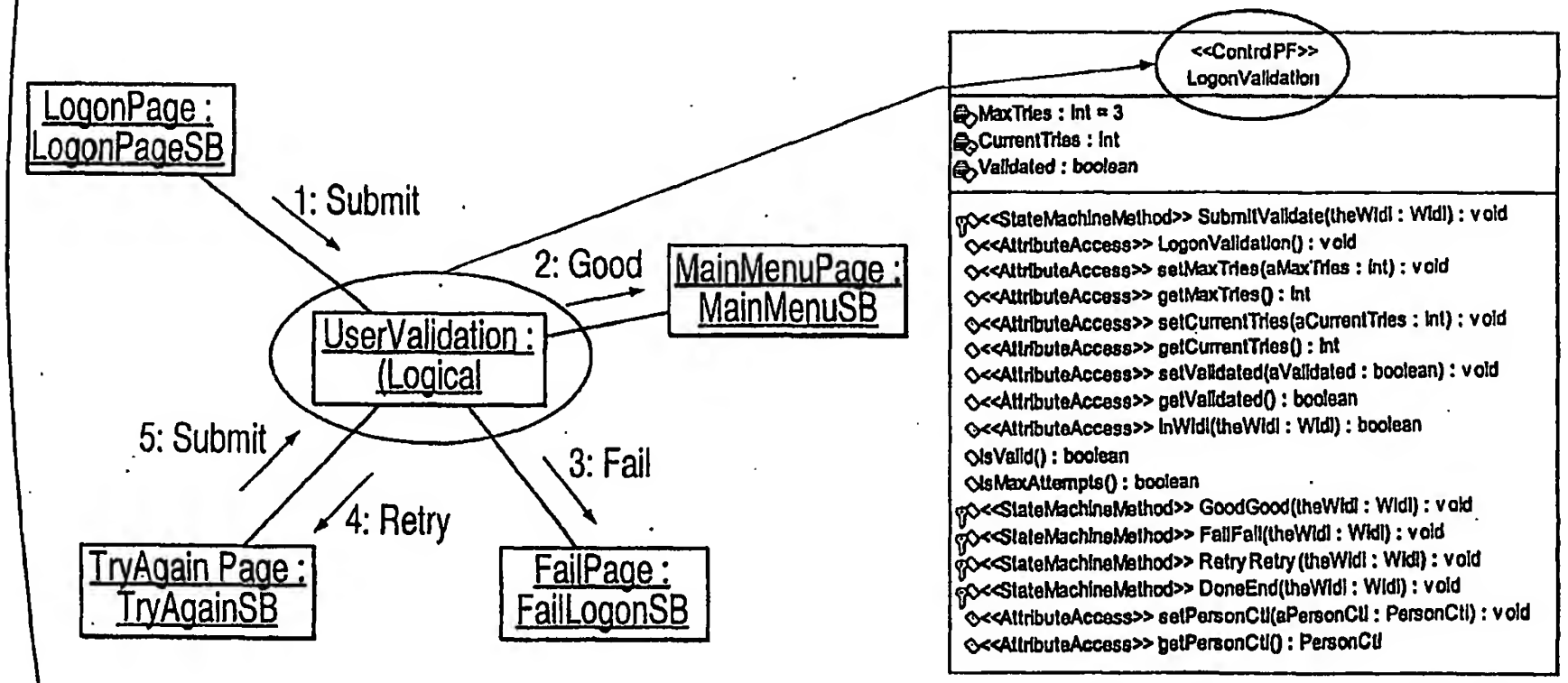


FIG. 47

Control Process Flow Class Specification

– The control process flow class specification provides the logical implementation making decisions through the use case collaboration process. The UML control process flow stereotyped class specification is generated into Java™ source code with all the static and dynamic behavior required for:

- Receiving input events
- Managing persistent data
- Manipulating Interface XML documents
- Returning information to the process.
- Maintaining state information of the application.

FIG. 48

Receiving Events

– When a user activates an event from an interface object, that event is sent into the state machine. All information coming into the state machine is contained within an XML document.

FIG. 49

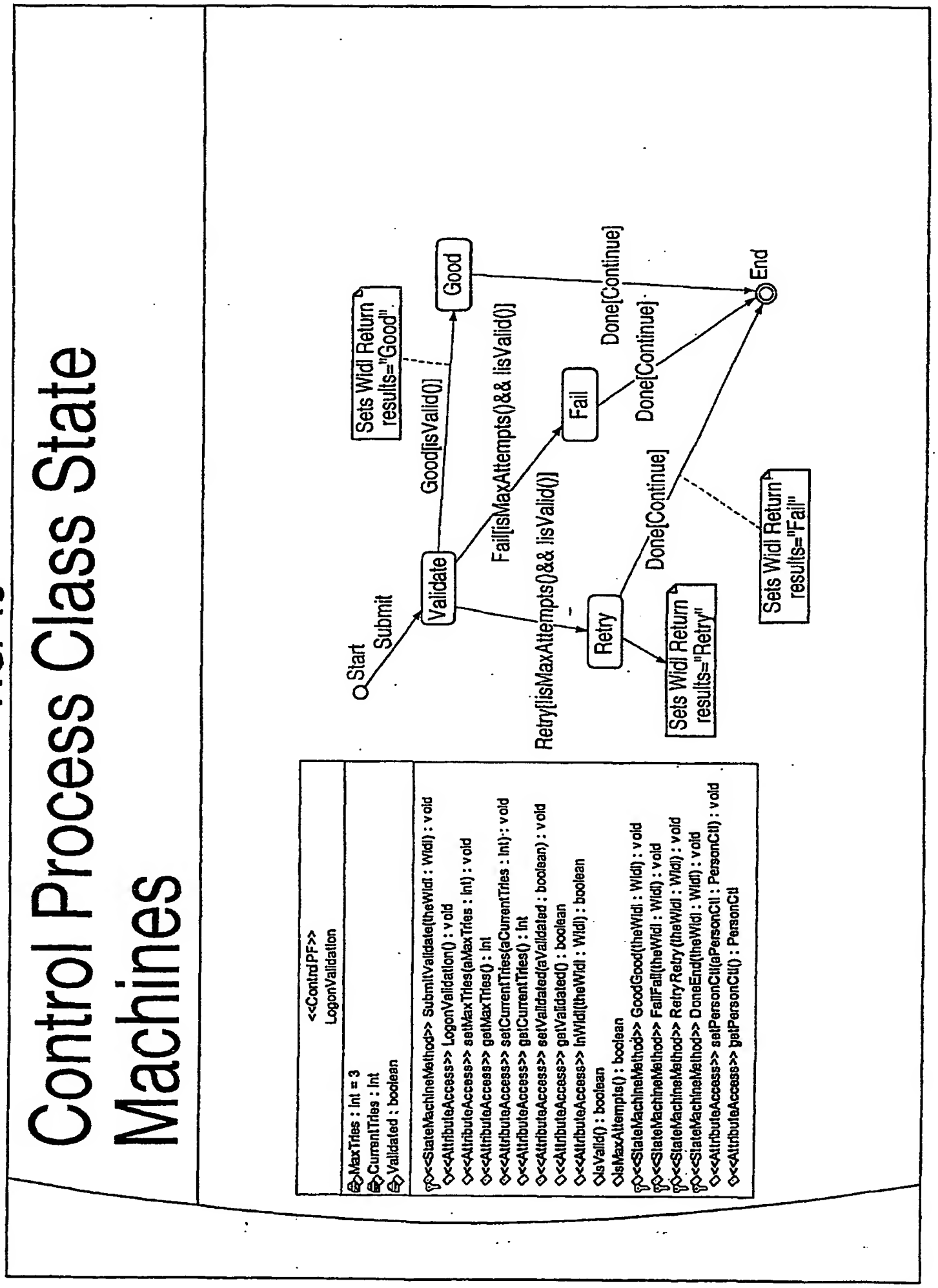

Control Process Class State Machines
Start
Submit
Validate
Retry[!isMaxAttempts()&& !isValid()]
Retry
Sets Widl Return results="Retry"
Fail[isMaxAttempts()&& !isValid()]
Fail
Sets Widl Return results="Fail"
Good[isValid()]
Good
Sets Widl Return results="Good"
Done[Continue]
Done[Continue]
Done[Continue]
End
<<ControlPF>>
LogonValidation
MaxTries : Int = 3
CurrentTries : Int
Validated : boolean
<<StateMachineMethod>> SubmitValidate(theWidl : Widl) : void
<<AttributeAccess>> LogonValidation() : void
<<AttributeAccess>> setMaxTries(aMaxTries : Int) : void
<<AttributeAccess>> getMaxTries() : Int
<<AttributeAccess>> setCurrentTries(aCurrentTries : Int) : void
<<AttributeAccess>> getCurrentTries() : Int
<<AttributeAccess>> setValidated(aValidated : boolean) : void
<<AttributeAccess>> getValidated() : boolean
<<AttributeAccess>> InWidl(theWidl : Widl) : boolean
IsValid() : boolean
IsMaxAttempts() : boolean
<<StateMachineMethod>> GoodGood(theWidl : Widl) : void
<<StateMachineMethod>> FailFail(theWidl : Widl) : void
<<StateMachineMethod>> RetryRetry(theWidl : Widl) : void
<<StateMachineMethod>> DoneEnd(theWidl : Widl) : void
<<AttributeAccess>> setPersonCtl(aPersonCtl : PersonCtl) : void
<<AttributeAccess>> getPersonCtl() : PersonCtl

FIG. 50

WIDL

- **WIDL is a Web Interface Definition Language specification.**
  - The Widl contains an event, Process, and structured records defining behavior and data together within a single XML document package.
  - Event or Method
    - The method described within the Widl is the event name coming into the state machine.
  - Process
    - The Widl process attribute identifies the executable Java class that will receive the Widl and respond to the method event. The process name can be a fully qualified Java class or an abstract object name. Containers receiving the Widl input from either session beans, JMS messaging Queue implementations etc., must have a mechanism to late bind the Widl event to the correct process.

FIG. 51

WIDL Records

The Widl contains three records for input information, output information and return information. These records can be populated with attributes or complete XML documents.

- Input Record
  - Our example for the Web application will use the input record to contain information coming from the Web interface into the process logic.
- Output Record
  - The example Widl use the output record of the Widl to contain Return Page or XML document information back to the Web interface
- Return Record
  - The Widl return record contains results information for evaluating decisions to the collaboration process flow. The result Attribute within the Widl Return record contains the return events from the control process flow. class specification.

FIG. 52

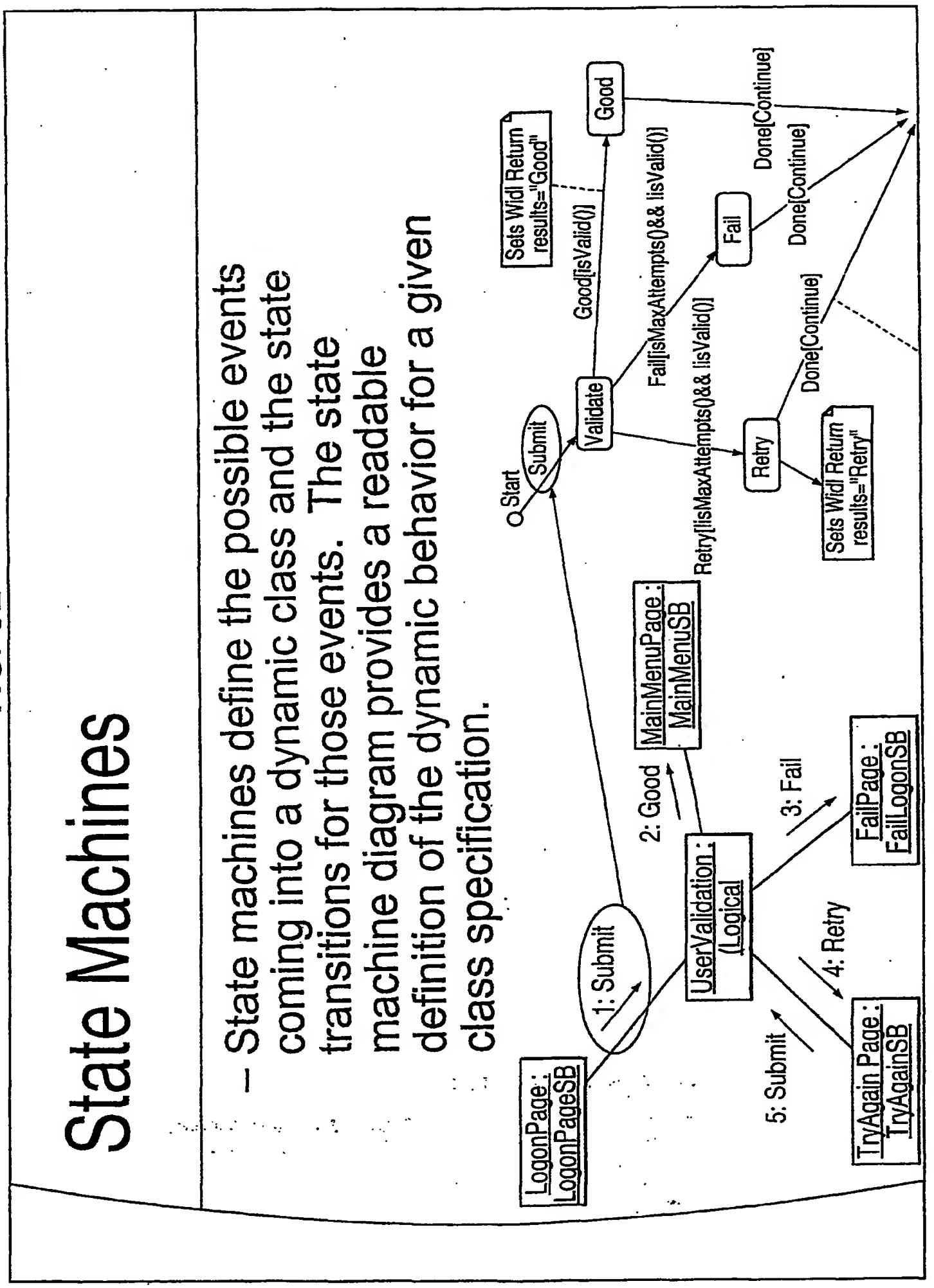

State Machines
– State machines define the possible events coming into a dynamic class and the state transitions for those events. The state machine diagram provides a readable definition of the dynamic behavior for a given class specification.
Start
Submit
Validate
Sets Widl Return results="Good"
Good[isValid()]
Good
Fail[isMaxAttempts()&& !isValid()]
Fail
Done[Continue]
Done[Continue]
Retry[!isMaxAttempts()&& !isValid()]
Retry
Done[Continue]
Sets Widl Return results="Retry"
LogonPage : LogonPageSB
1: Submit
UserValidation : (Logical
2: Good
MainMenuPage : MainMenuSB
3: Fail
FailPage : FailLogonSB
4: Retry
5: Submit
TryAgain Page : TryAgainSB

FIG. 53

States and Transitions

- *States*
  - States define the process stops through the state machine.
- *Event Transitions*
  - Transitions define the event causing a transition from one state to another state.
- *Event State Methods*
  - Event state methods are the implied action behaviors when an event transition occurs. These event state methods are implemented through UML activity diagrams.

FIG. 54

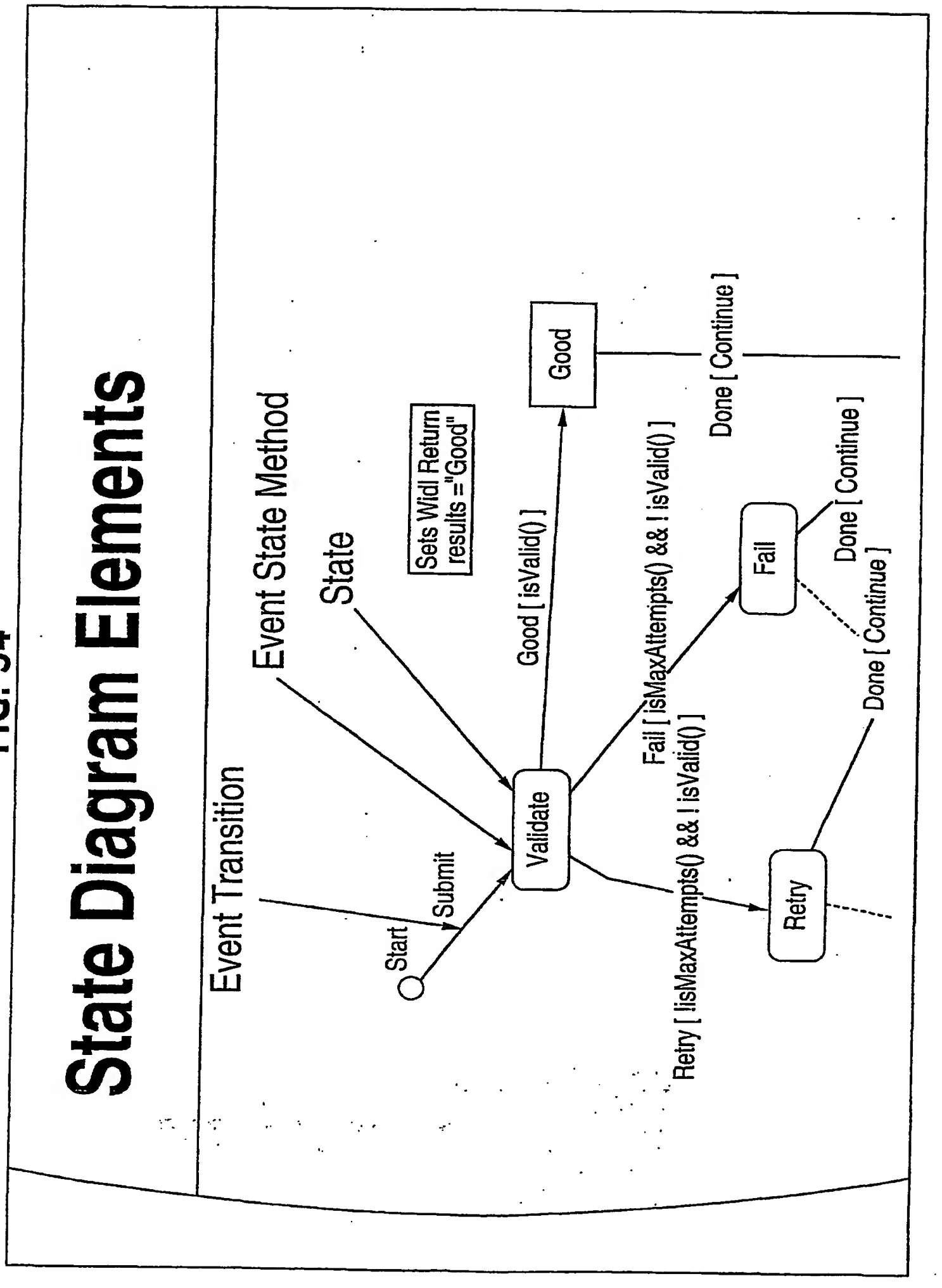

State Diagram Elements
Event State Method
State
Sets Widl Return results = "Good"
Good
Done [Continue]
Good [isValid()]
Event Transition
Start
Submit
Validate
Fail [isMaxAttempts() && ! isValid()]
Fail
Done [Continue]
Retry [!isMaxAttempts() && ! isValid()]
Retry
Done [Continue]

FIG. 55

State Timing

- *Guard Conditions*
  - Guard conditions imply a synchronous transition exiting a given state. The guard conditions contain boolean logic to determine which exit transition will be traversed.
- *Asynchronous*
  - Asynchronous events have no guard conditions and are triggered from some external source. That source is typically a user interface link artifacts.
- *Synchronous*
  - Synchronous events transition automatically and internal to the state machine. Synchronous events are identified as exit events from a state containing guard conditions.

WO 01/08007
58/76
PCT/US00/20069

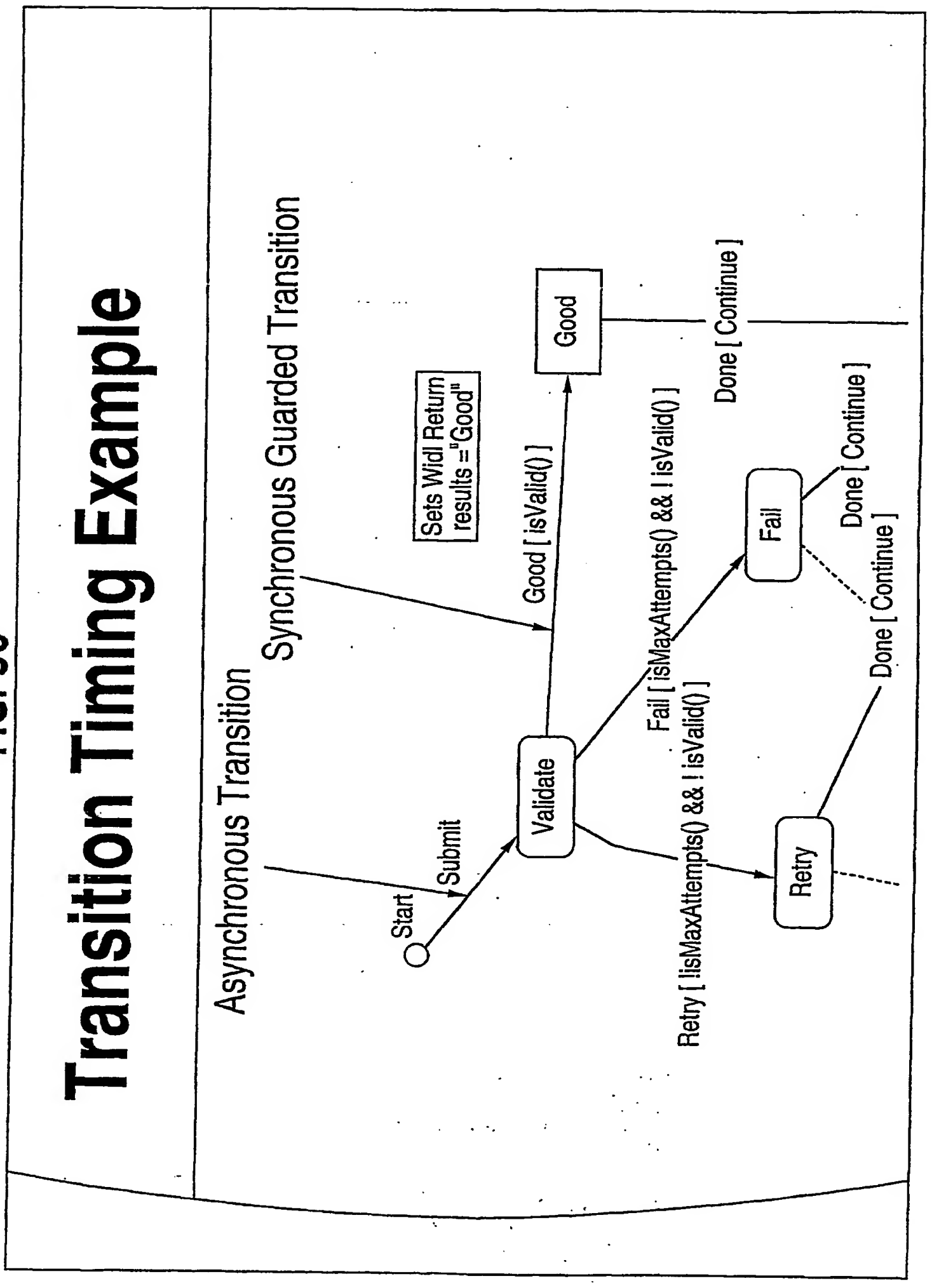

Example
Transition Timing Example
Asynchronous Transition
Synchronous Guarded Transition
Start
Submit
Validate
Retry [!isMaxAttempts() && ! isValid()]
Fail [isMaxAttempts() && ! isValid()]
Retry
Done [Continue]
Sets Widl Return results ="Good"
Good [isValid()]
Good
Done [Continue]
Fail
Done [Continue]

FIG. 56

FIG. 57

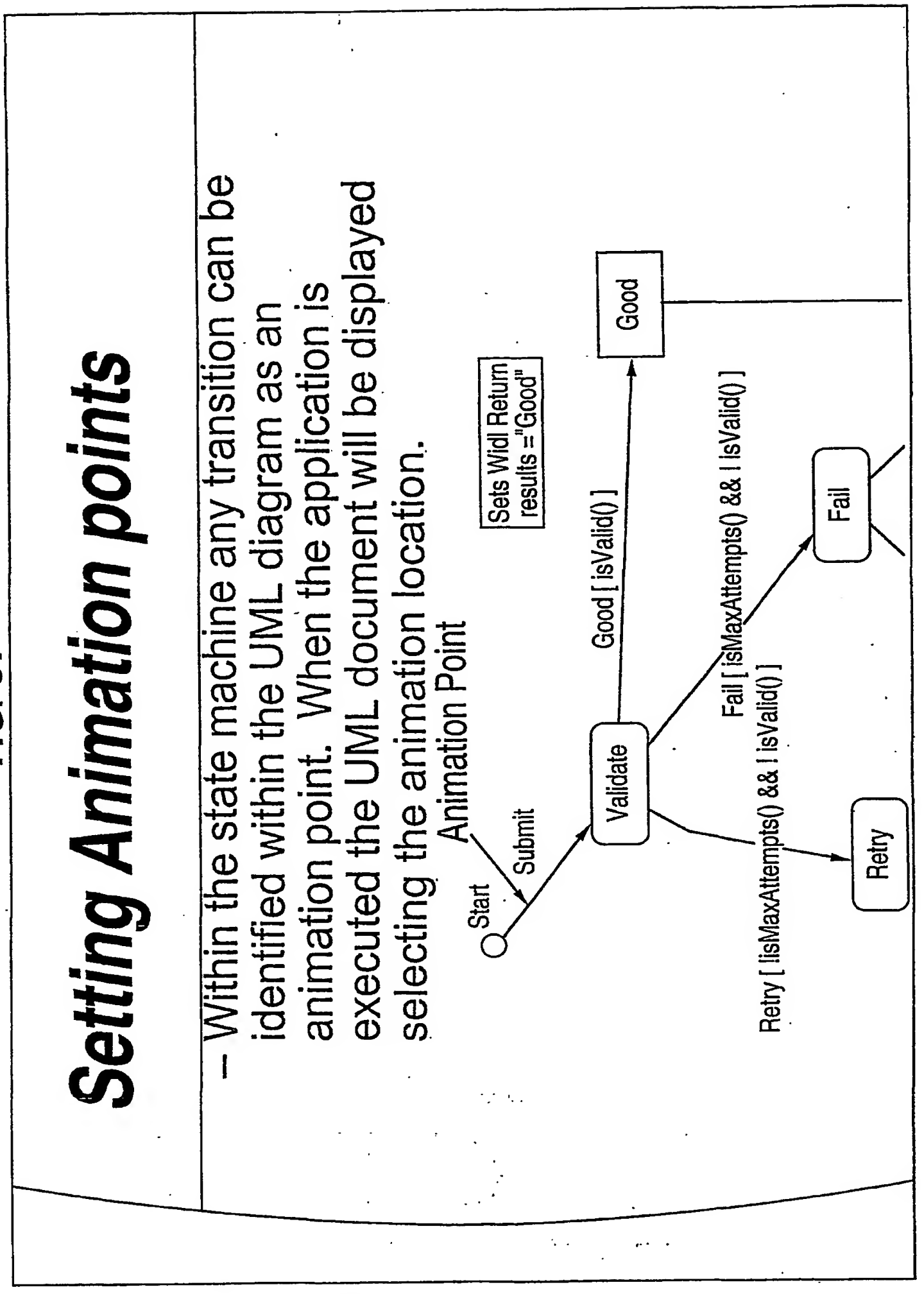

Setting Animation points
– Within the state machine any transition can be identified within the UML diagram as an animation point. When the application is executed the UML document will be displayed selecting the animation location.
Animation Point
Start
Submit
Validate
Sets Widl Return results ="Good"
Good [isValid()]
Good
Fail [isMaxAttempts() && ! isValid()]
Fail
Retry [!isMaxAttempts() && ! isValid()]
Retry

FIG. 58

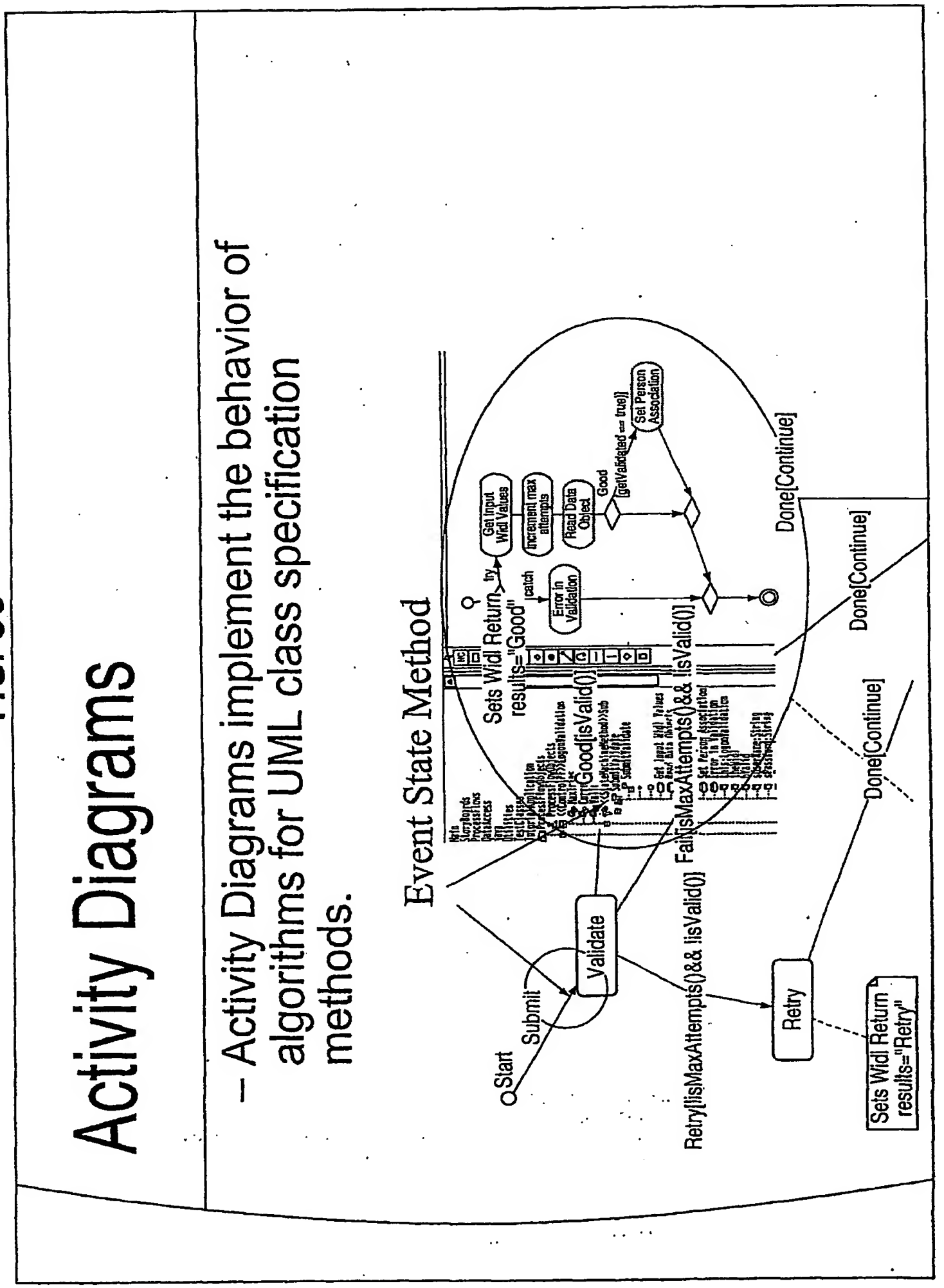

Activity Diagrams
– Activity Diagrams implement the behavior of algorithms for UML class specification methods.
Event State Method
Start
Submit
Validate
Retry[IisMaxAttempts()&& !isValid()]
FailIsMaxAttempts()&& !isValid()]
Retry
Sets Widl Return results="Retry"
Done[Continue]
Sets Widl Return results="Good"
Error in Validation
Get Input Widl Values
Increment max attempts
Read Data Object
Good [getValidated == true]
Set Person Association
Done[Continue]

FIG. 59

UML Object Navigation Notation

– Activity Diagrams are represented in UML Object Navigation Notation. This notation manipulates the objects with their public attributes and methods. Using the object notation and Activity Editor within UML Factory you can define the complete behavior for a method implementation.

```
sUserName = theWidl.getInputParameter( "UserName" )
sPassWord = theWidl.getInputParameter( "Password" )
this.setValidated( false )
```

FIG.60

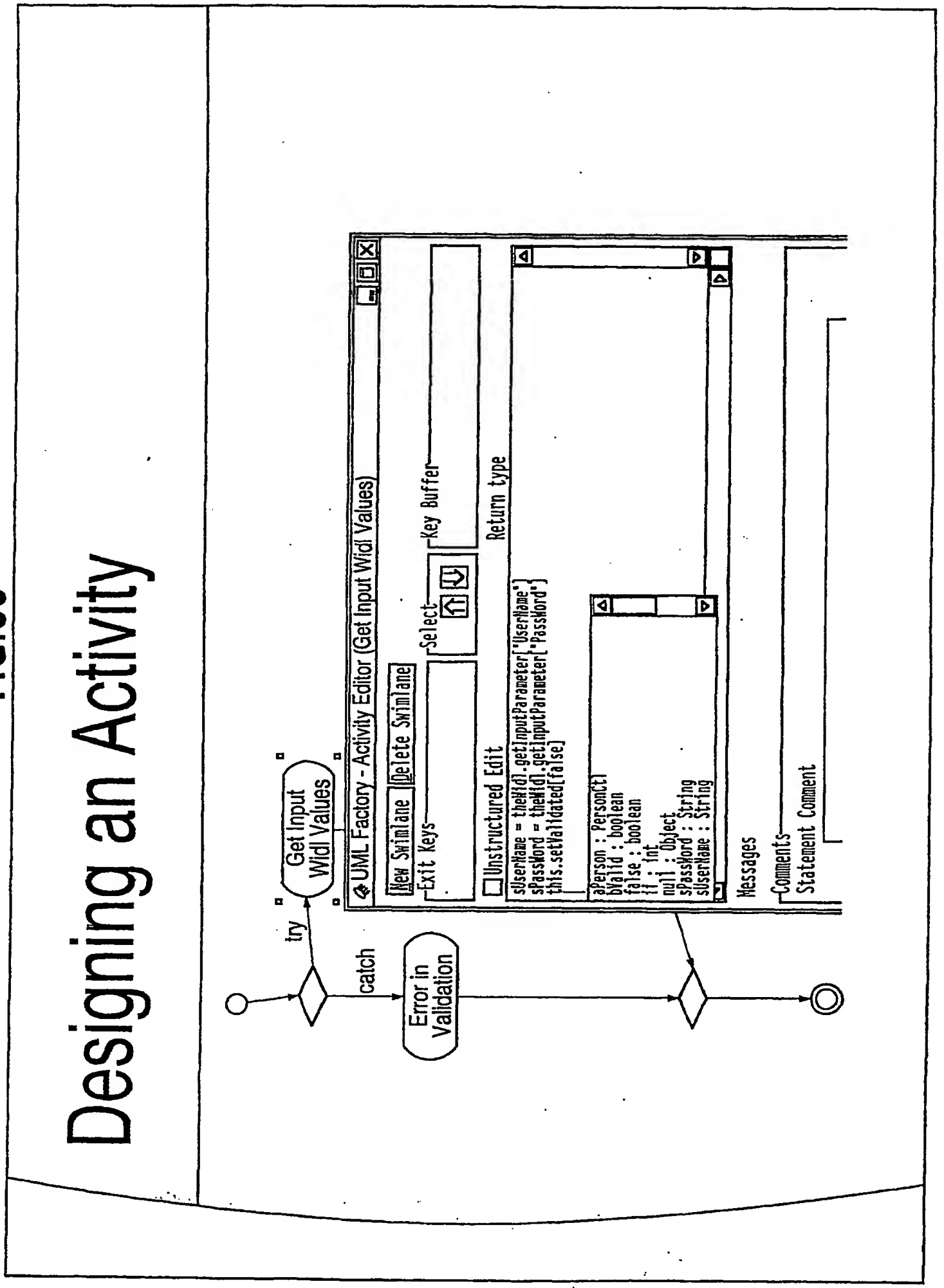

Designing an Activity
Get Input Widl Values
try
catch
Error in Validation
UML Factory - Activity Editor (Get Input Widl Values)
New Swimlane
Delete Swimlane
Exit Keys
Select
Unstructured Edit
sUserName = theWidl.getInputParameter["UserName"]
sPassWord = theWidl.getInputParameter["PassWord"]
this.setValidated[false]
aPerson : PersonCt
bValid : boolean
false : boolean
ii : int
null : Object
sPassWord : String
sUserName : String
Messages
Comments
Statement Comment
Widl Values)
Key Buffer
Return type

FIG. 61

Accessing a Data Base

- Through the activity diagram implementations we can manipulate our database objects, defining the operations for all our database elements as required.

```
aPerson.PersonCtl( )
aPerson.setFirstName( sUserName )
aPerson.setPassword( sPassWord )
this.setValidated( aPerson.load( ) )
```

FIG. 62

Manipulating XML

– Within the activity diagrams we also define the XML manipulation for retrieving and building the XML documents going between the process flow logic and user interface elements.

```
sValue= new String( "PassageSoftware" );
aXMLMap.insert( this.getXML_CompanyName( ), sValue );
sValue= new String( "Richmond, VA" );
aXMLMap.insert( this.getXML_CompanyCity( ), sValue );
aXMLMap.insert( this.getXML_Person_NODE( ), null );
sName= new String( "Dick" );
sLName= new String( "Douglas" );
aXMLMap.insert( this.getXML_FirstName( ), sName );
aXMLMap.insert( this.getXML_LastName( ), sLName );
```

FIG. 63

Intelligent Advisor Code view

— The UML Factory Intelligent Adviser allows us to view the code generation results while designing the UML activity diagrams.

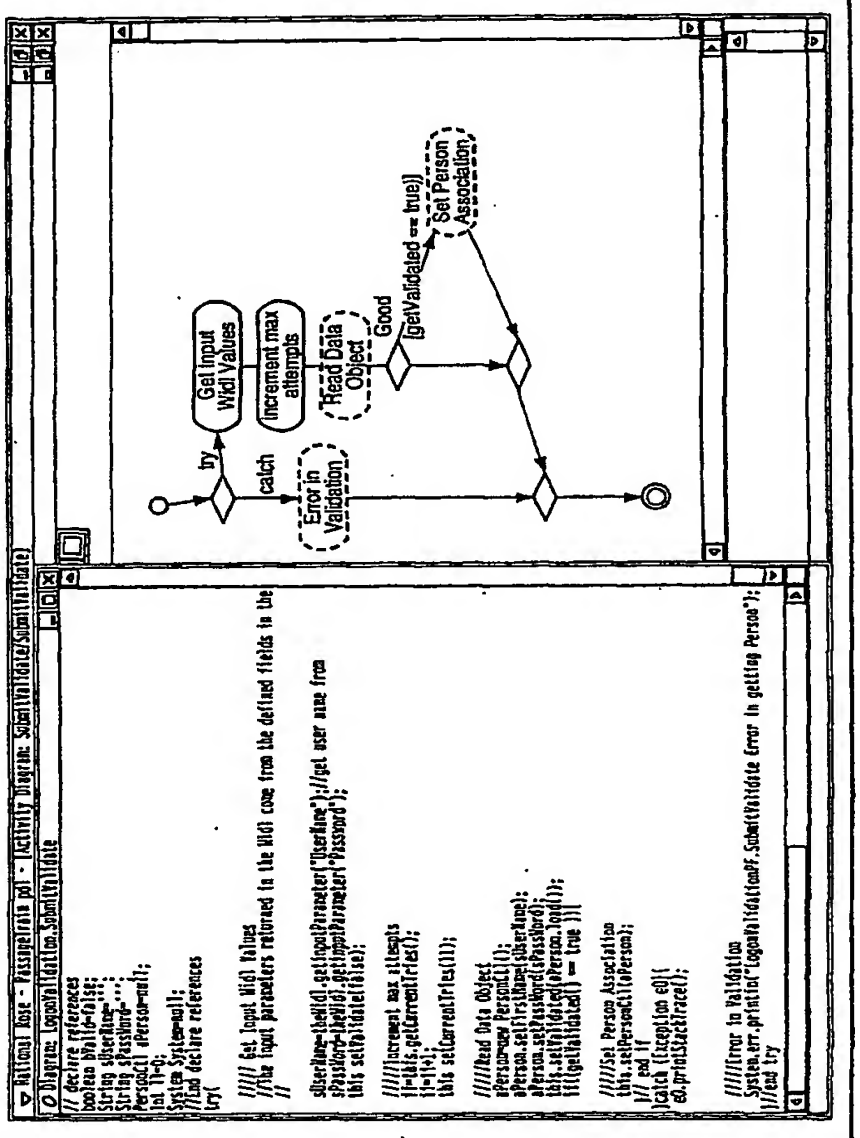
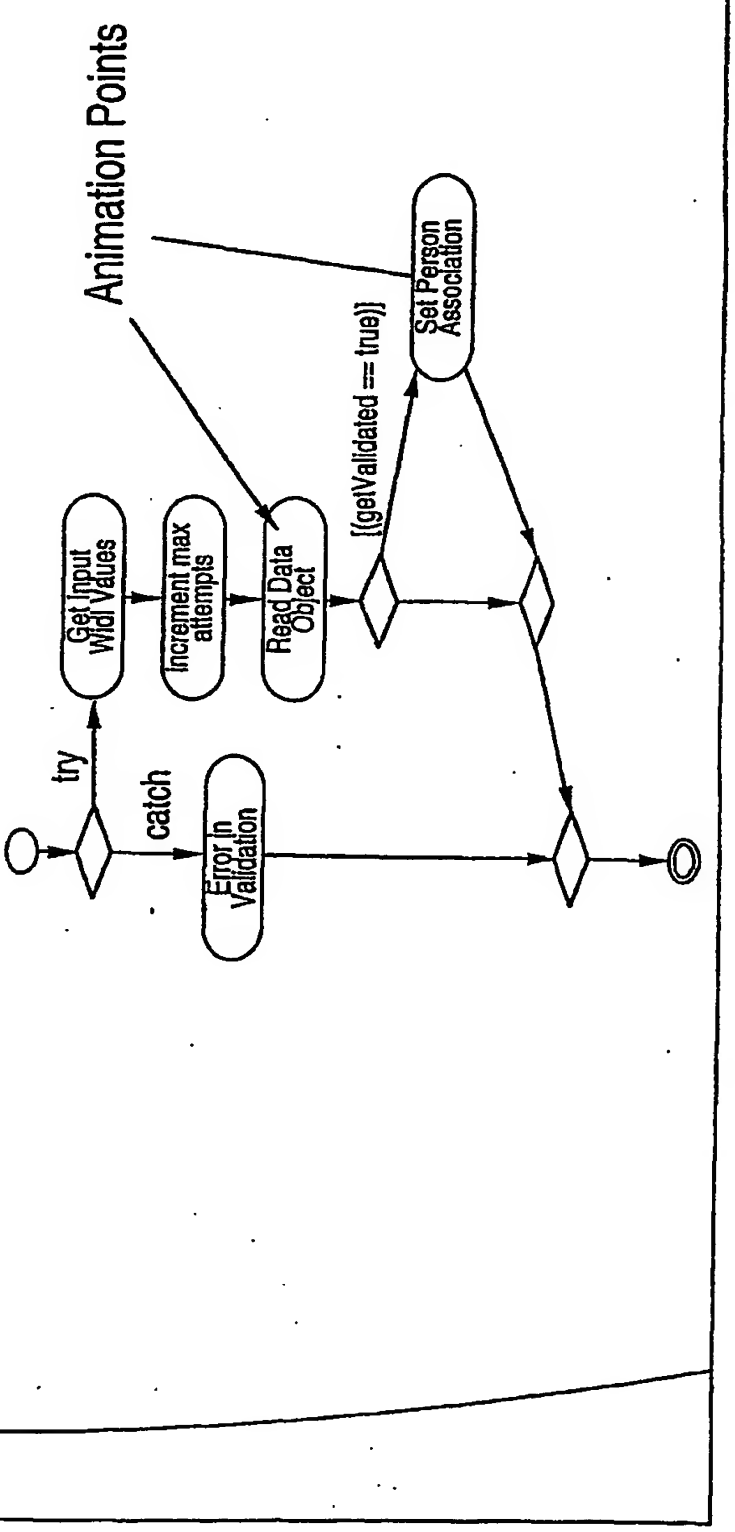


FIG. 64

Setting Animation points

– Any activity within the activity diagrams can also have animation points defined. When the application is executing the activity diagram will display and select the activities with animation settings.



Generating the Application

– After the static and dynamic elements of our application are defined we can organize these elements into UML Component Diagrams to define deployment definitions. From the components definitions we can then generate the complete Java class code, compile the code and package it into the appropriate application JAR.

FIG. 65

FIG. 66

Collaboration Diagram XML Generation

– The collaboration diagram information for our use case definitions is also generated into an XML description document identifying the interface elements, XML mappings, and control process flow information. This XML document then becomes the basis for execution through the application.

- The XML document defines the process flow through the application.
- The engine that iterates the collaboration diagram XML document was also designed and generated from a UML model.

FIG. 67

Control Process Flow Generation

– Dynamic behavior represented within an application is designed within a control process flow class specification. The following sections will briefly illustrate the generation processing capabilities for dynamic behavior within a UML environment. The following code examples were completely generated from the UML model.

PCT/US00/20069

FIG. 68

Generating the UML Class Specification

- *Looking at the State Machine implementation*
  - Current State
  - Incoming Event
  - Synchronous Transitions
  - Asynchronous Transitions
  - Start and End States
- *Packages, imports, declarations*
  - Java package specifications can be explicitly or implicitly defined by the class locations within the UML model.
- *Association and Attribute implementations*
  - Association implementations generate both the container and access methods for the stereotyped association. The type of container and resulting access methods are defined by the stereotype applied to the association between class specifications.

FIG. 69

Method Implementation

– *Activity Diagram Method Implementations*

- Activity diagrams are generated following the UML Object Notation. The activity generation can be viewed with either as a fully generated class or using the Intelligent Advisor code window.

– *Round Trip engineering*

- Roundtrip engineering allows developers to hand edit code generated from the UML Diagram. By Identifying elements which have been hand edited, the UML Factory generator will leave the method implementation as edited without forward generating from the model. The developer controls the granularly of roundtrip engineering.

– *UML Model Animation code*

- Animation points placed within the code can be turned off for individual points, the entire model being generated and also ignored, from run time configuration properties

Executing the Architectures

– The following demonstrations will walk through executing the various architectures from the complete components on different types of application servers.

FIG. 70

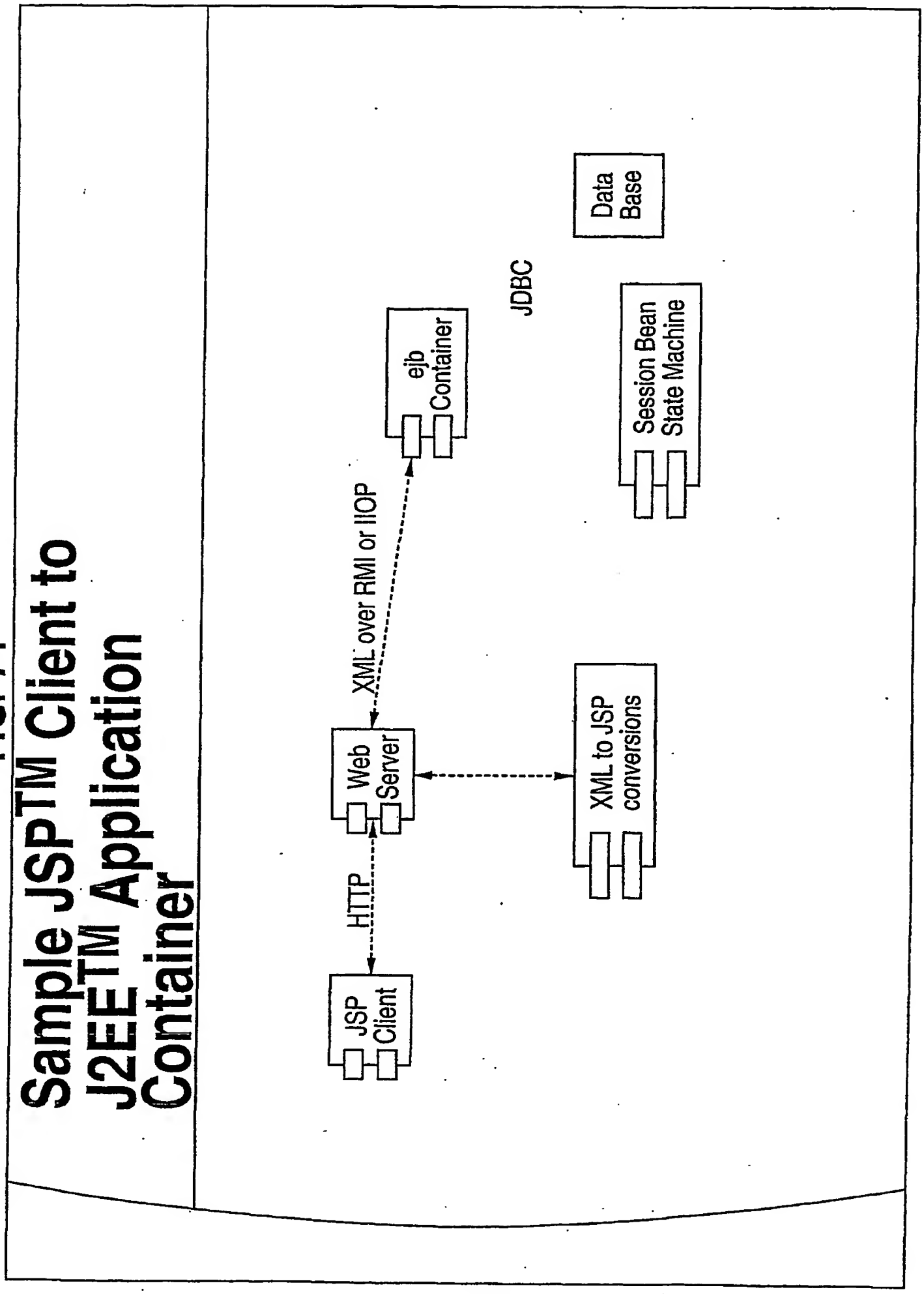

Sample JSP™ Client to J2EE™ Application Container
Data Base
JDBC
ejb Container
Session Bean State Machine
XML over RMI or IIOP
Web Server
XML to JSP conversions
HTTP
JSP Client

FIG. 71

FIG. 72

Summary

- This concludes the presentations and demonstrations for building n-tier enterprise applications with UML and XML data representations.
- The demonstrations have illustrated the ability to completely model both static and dynamic application behavior within a UML model.
- The UML model also defined components and deployment into multiple configurations.
- The modeled demonstration illustrated the ability to use the same components within different J2EE™ platform-enabled implementations on multiple architectural environments.

FIG. 73

Enterprise Development Benefits

- Enterprise computing can improve the software development process by combining the descriptive power of the UML notation, and the flexible data representation and distribution capabilities of XML with the object oriented, network, and portable capabilities of Java technology.
- Combining these technologies enables organizations to manage components increasing reuse, visibility, and implementation understanding; effectively reducing the complexity and total cost of ownership for deploying n-tier enterprise applications

INTERNATIONAL SEARCH REPORT

Internal Application No
PCT/US 00/20069

A. CLASSIFICATION OF SUBJECT MATTER
IPC 7 G06F9/44

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 7 G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)
EPO-Internal, INSPEC, IBM-TDB

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	US 5 455 952 A (GJOVAAG INGHARD J) 3 October 1995 (1995-10-03) column 2, line 61 -column 3, line 13 column 10, line 23 - line 36 -/--	1-13

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
28 November 2000	05/12/2000

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Brandt, J

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Intern al Application No
PCT/US 00/20069

C.(Continuation) DOCUMENTS CONSIDERED TO BE RELEVANT

Category °	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
A	TOSHIMI MINOURA ET AL: "STRUCTURAL ACTIVE OBJECT SYSTEMS FOR SIMULATION" ACM SIGPLAN NOTICES,US,ASSOCIATION FOR COMPUTING MACHINERY, NEW YORK, vol. 28, no. 10, 1 October 1993 (1993-10-01), pages 338-355, XP000411736 ISSN: 0362-1340 page 340, left-hand column, line 32 -right-hand column, line 2 page 341, left-hand column, line 24 -right-hand column, line 35 page 352, right-hand column, line 5 - line 15 ---	1-13
A	WO 97 35254 A (MASSACHUSETTS INST TECHNOLOGY) 25 September 1997 (1997-09-25) page 2, line 6 -page 6, line 17 -----	1-13

Form PCT/ISA/210 (continuation of second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Intern al Application No
PCT/US 00/20069

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
US 5455952 A	03-10-1995	NONE	
WO 9735254 A	25-09-1997	CA 2249386 A	25-09-1997
		EP 0888585 A	07-01-1999